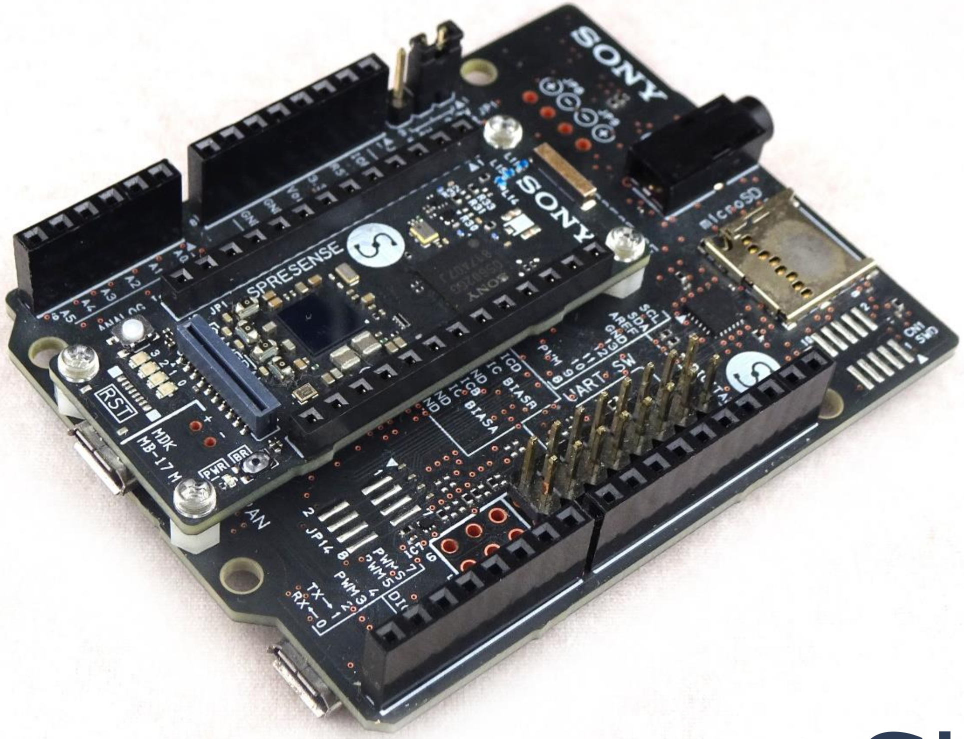
Get started Real-time Signal Processing with Sony SPRESENSE™

Sony Semiconductor Solutions Corporation
Yoshinori Oota

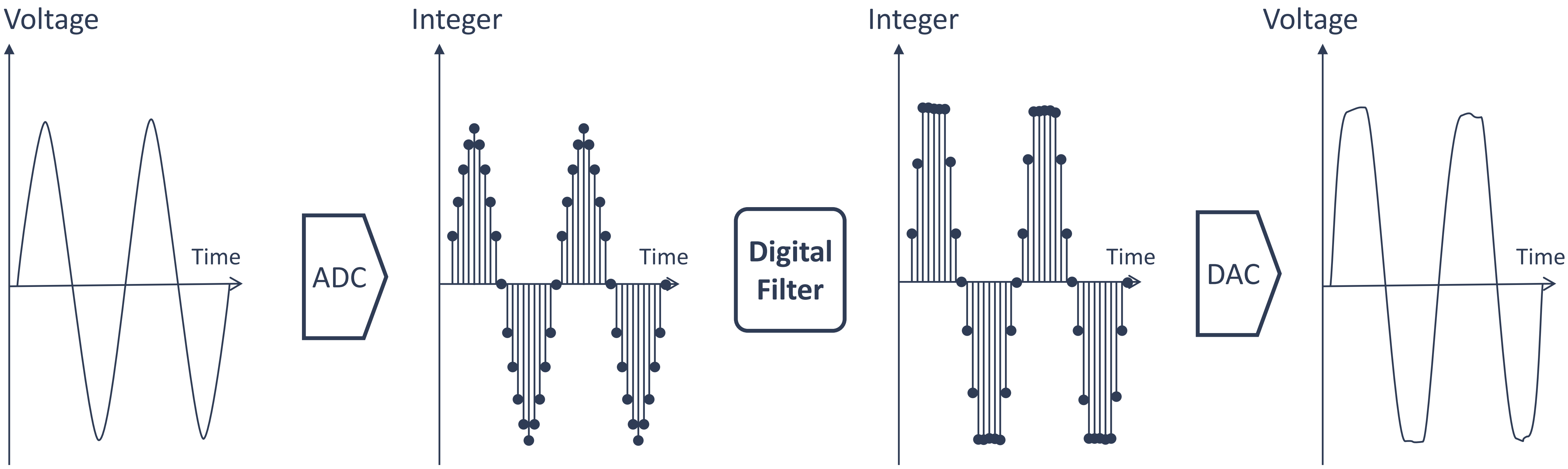
Contents



SPRESENSE

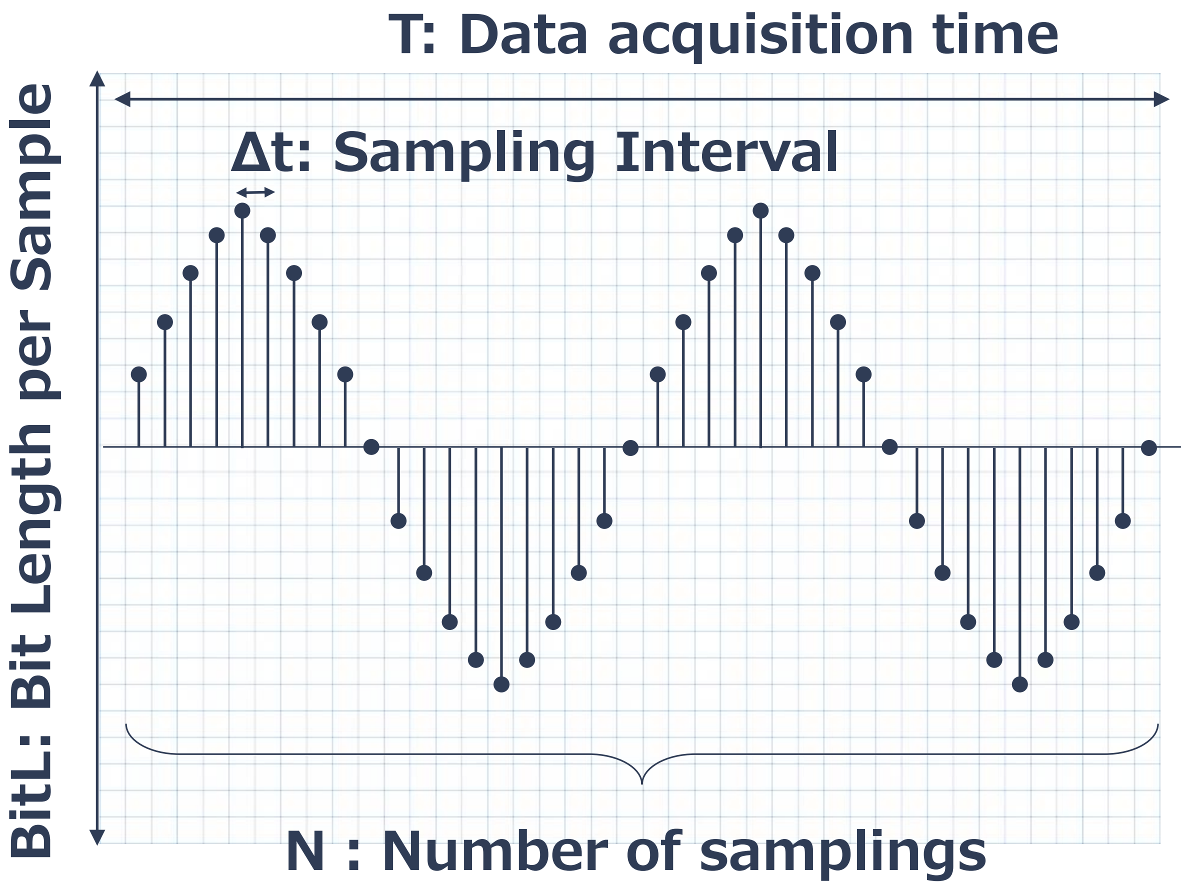
Signal Processing Basic

Real-time Digital Signal Processing



Real-time signal processing must get through from analog input to analog output in a limited short time

Parameters of Digital Signal Processing



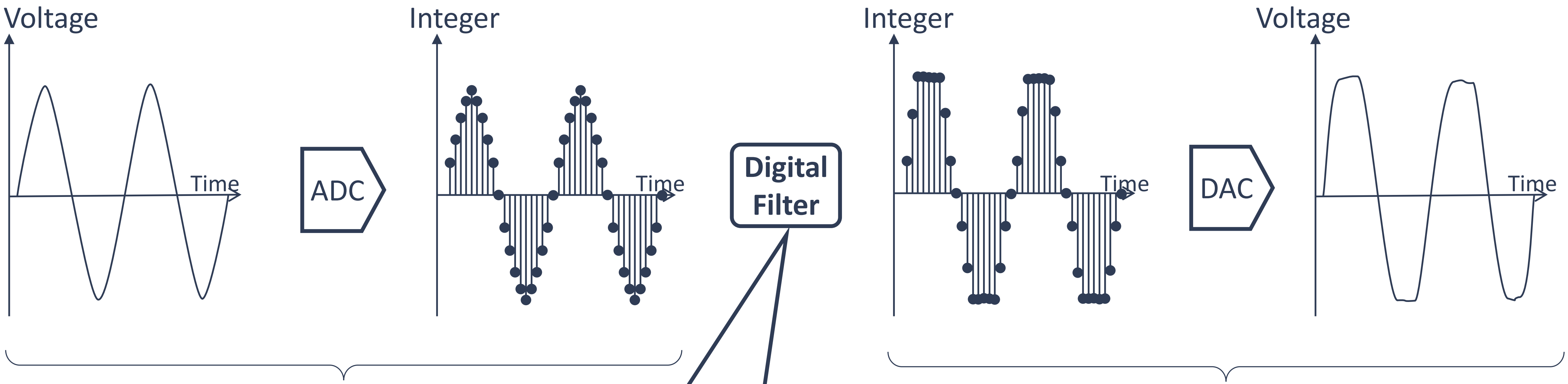
Symbol	Description	Memo	Unit
f_s	Sampling Frequency	$f_s = 1/\Delta t$	Hz
Δt	Sampling Interval	$\Delta t = 1/f_s$	Sec
N	Number of Samplings	$N = T/\Delta t = Tf_s$	-
T	Data acquisition time	$T = N/f_s$	Sec
BitL	Bit Length per sample	16 or 24Bits	V

Processing time is determined by sampling frequency and number of data. And the processing frequency is limited to 1/2 the sampling frequency (sampling theorem)

Sampling frequency: Number of data to be digitally converted per second

Number of samplings: Number of data to be stored in a buffer

Parameters of Digital Signal Processing



The time required for AD conversion is determined by “the number of data / sampling frequency”.

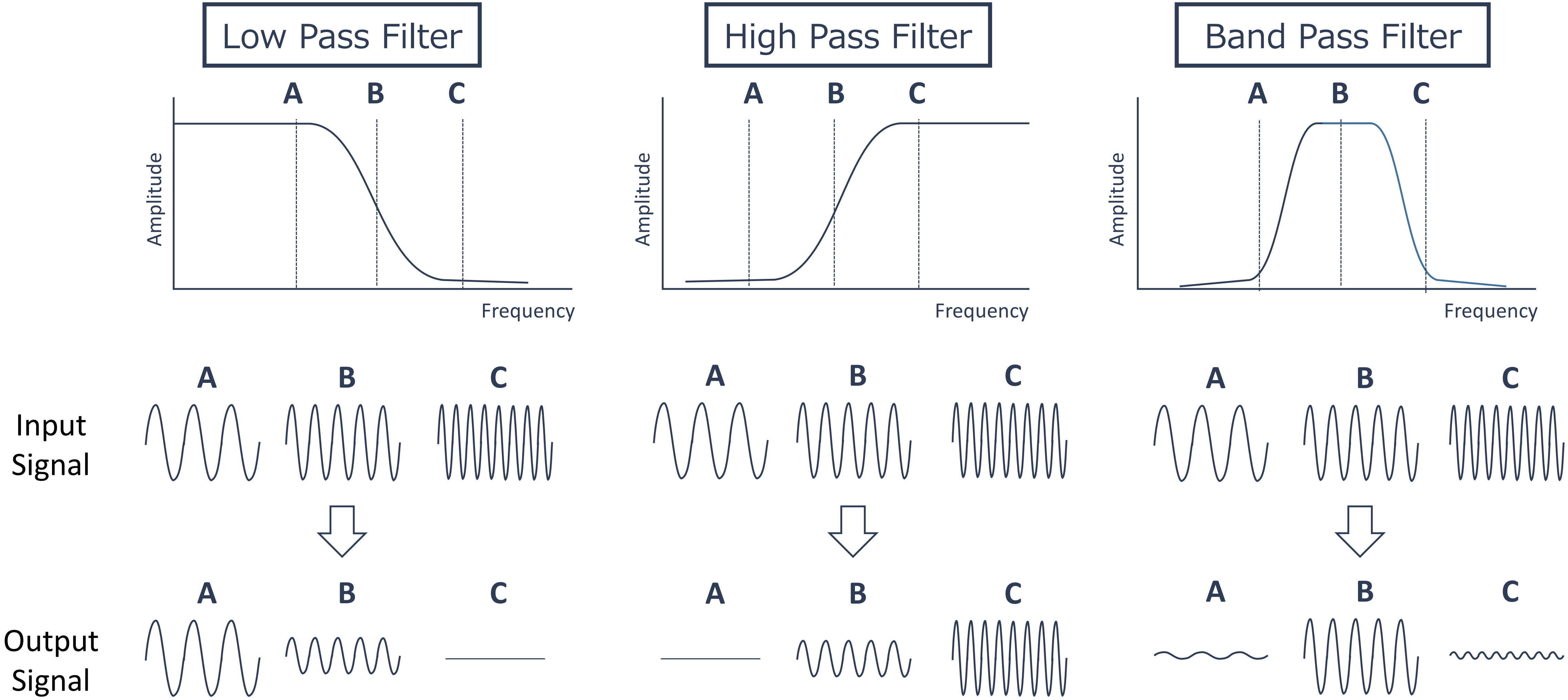
Example)

Number of data: 1024 (pcs)
Sampling frequency: 48000 (Hz)
Time for AD conversion: 0.0213 (sec)

The calculation time applied to the digital filter is the "<u>number of data / sampling frequency</u>”

Since DAC is processed by Hardware, the processing time is not a major consideration

Digital Filter Type

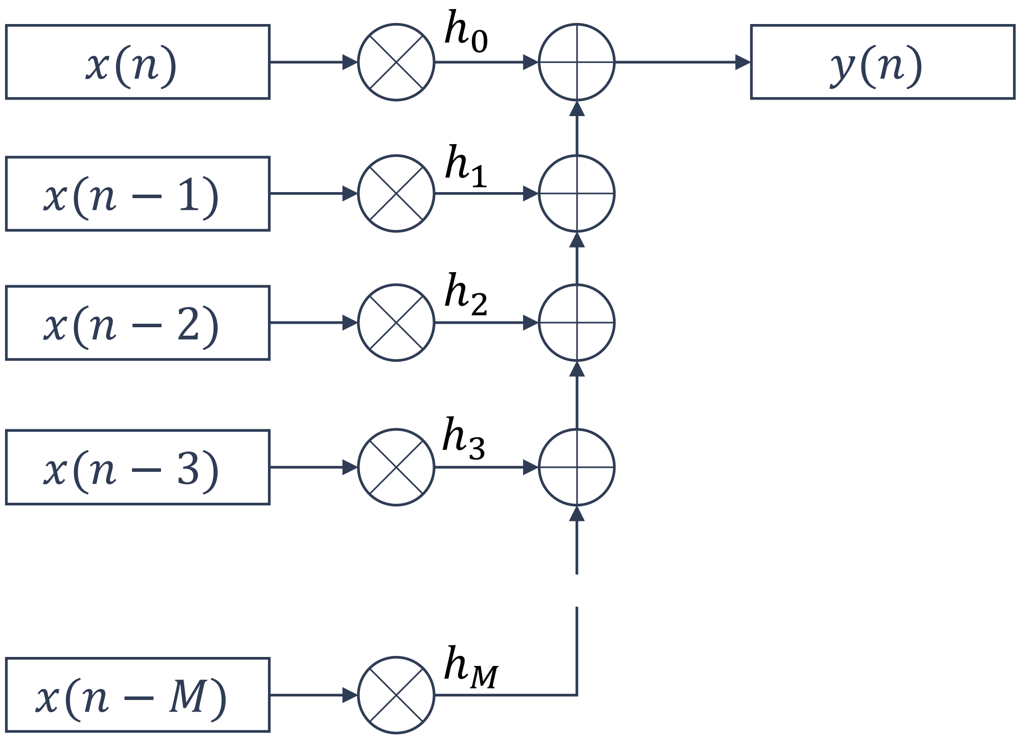


Digital Filter Type

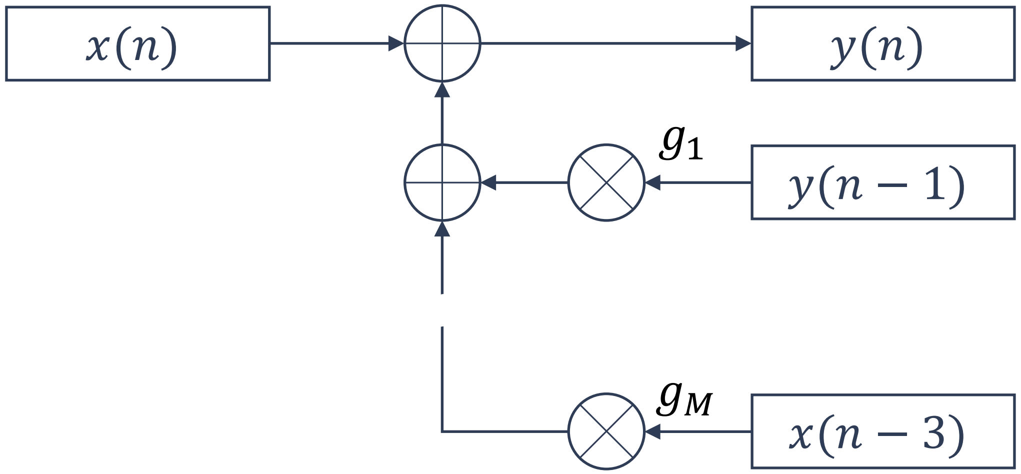
Type	Pros	Cons
FIR Finite Impulse Response filter	✓ Low phase distortion ✓ Structurally stable (no oscillation)	✓ Needs lots of calculation resources ✓ Relatively large latency
IIR Infinite Impulse Response filter	✓ Small calculation resources ✓ High-sped processing ✓ Low latency	✓ Phase distortion ✓ Possible of oscillation
STFT Short Time Fourier Transform	✓ Possible of wide variety of processing	✓ Needs high-end micro-processor due to complex processing ✓ Needs lots of calculation resources ✓ Large latency

Structure of FIR and IIR Filter

FIR $$y(n) = \sum_{m=0}^{M} h_m x(n-m)$$



IIR $$y(n) = x(n) + \sum_{m=1}^{M} g_m y(n-m)$$



IIR filter is a feedback structure

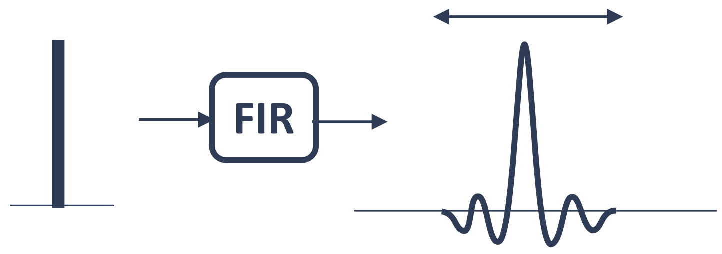
Characteristics of FIR and IIR filter

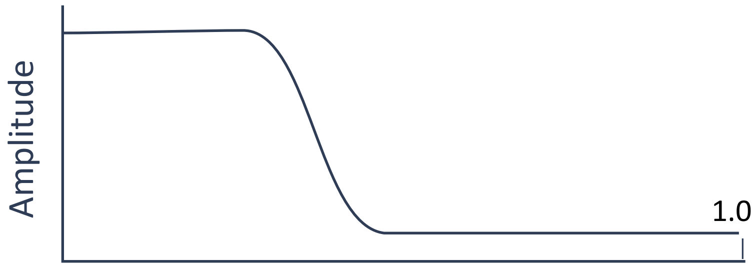
Impulse response and amplitude characteristics of FIR and IIR filters

FIR

Impulse response

Finite Response





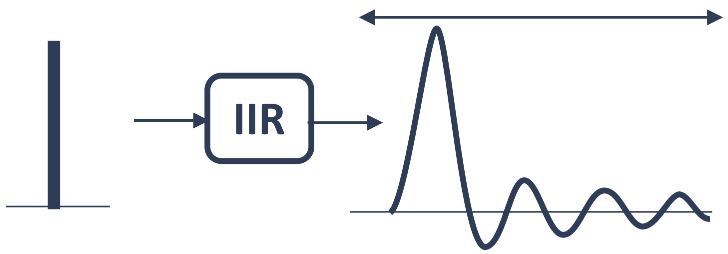
Normalized Frequency

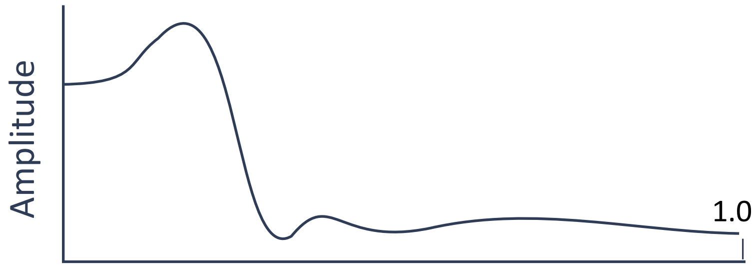
Sampling frequency normalized to 1.0

IIR

Impulse response

Infinite Response





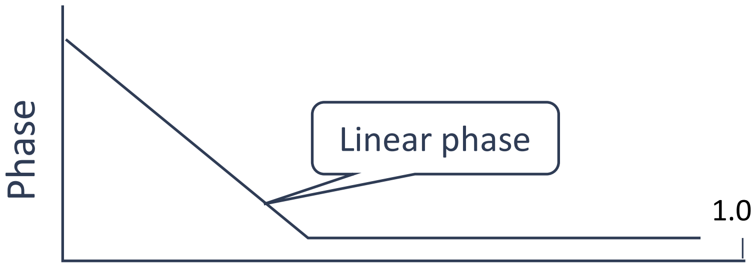
Normalized Frequency

Sampling frequency normalized to 1.0

Characteristics of FIR and IIR filter

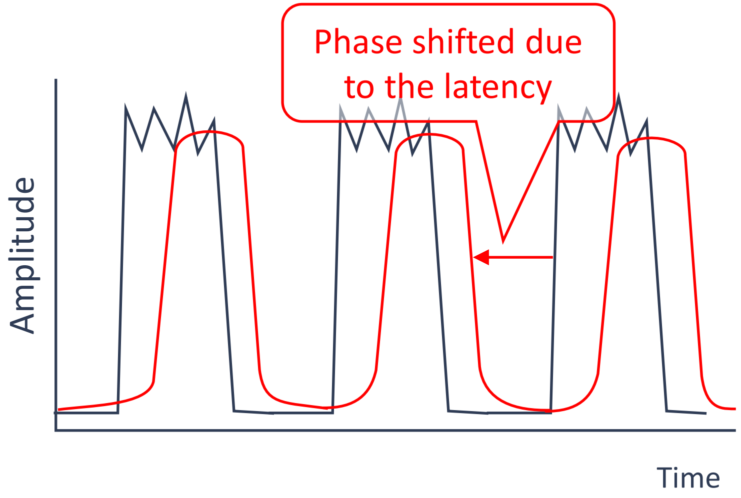
Phase characteristics of FIR and IIR filters

FIR



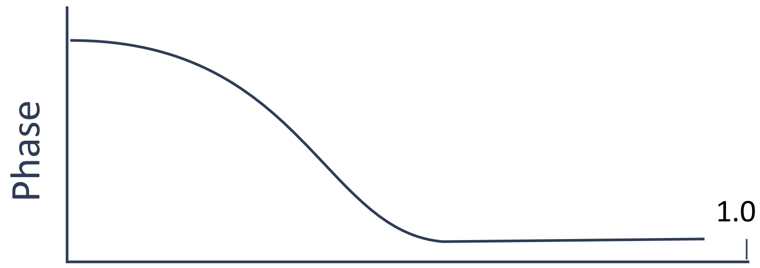
Normalized Frequency

Sampling frequency normalized to 1.0



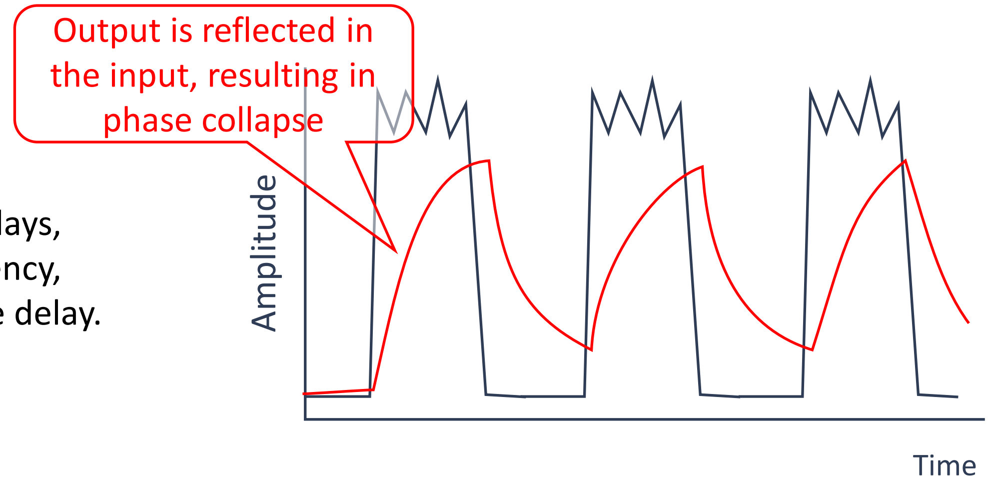
Due to processing delays, the higher the frequency, the greater the phase delay.

IIR

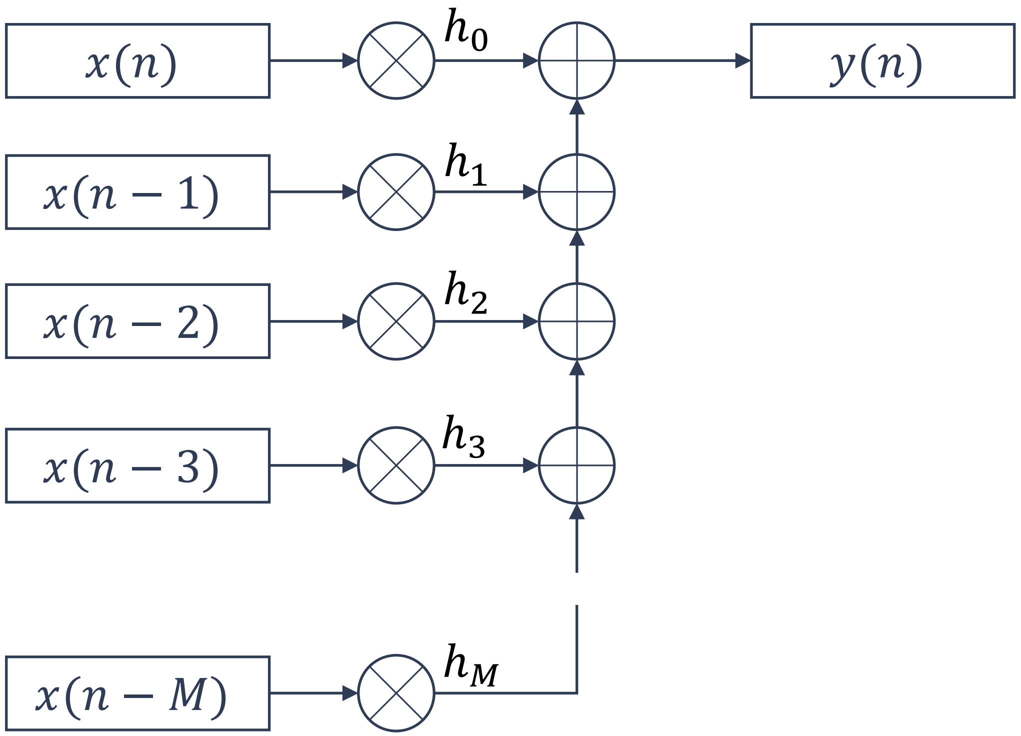


Normalized Frequency

Sampling frequency normalized to 1.0



FIR Low Pass Filter



Equation of Coefficients for FIR Low Pass Filter

F_c : Cutoff Frequency
F_s : Sampling Frequency
$f_c = \frac{F_c}{F_s}$: Normalized Cutoff Frequency

$$h_{k+M/2} = \begin{cases} 2f_c & k = 0 \\ 2f_c \dfrac{\sin(2\pi f_c k)}{2\pi f_c l} & k \neq 0 \end{cases}$$

$-\frac{M}{2} \leq k \leq \frac{M}{2}$
k is integer

Apply a window function (Han window in this case) to suppress the ripple generated by the frequency response of the filter

$$h_m = w_m h_m$$

$$w_m = 0.5 - 0.5cos\left(\frac{2\pi m}{M}\right) \quad m = 0 \ldots M$$

FIR Low Pass Filter

Example

F_c=2000(Hz) : Cutoff Frequency

F_s=48000(Hz) : Sampling Frequency

$f_c = \frac{F_c}{F_s}$=0.041667 : Normalized Cutoff Frequency

M=63 : Taps

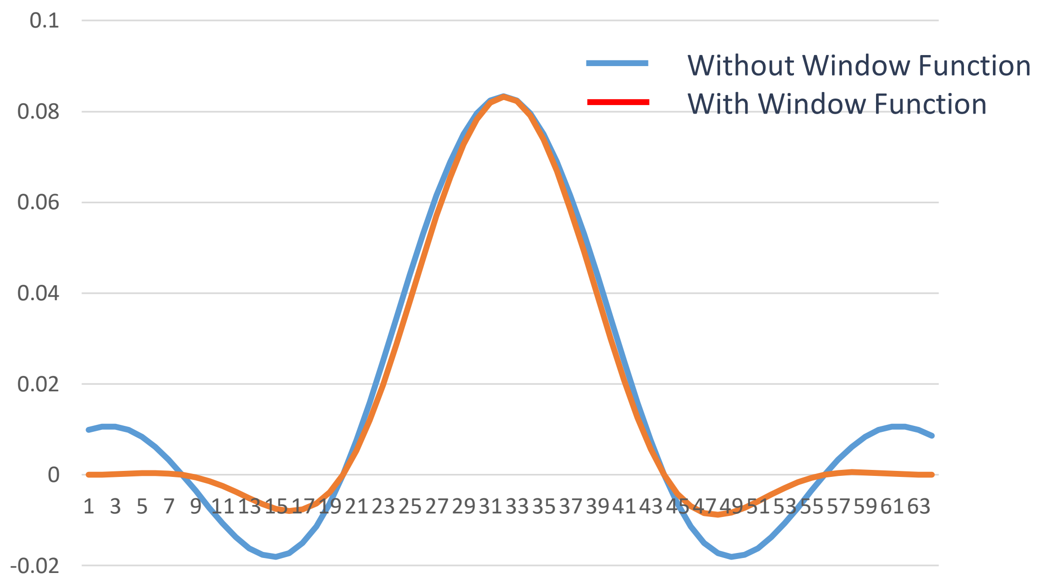
$$h_{k+M/2} = \begin{cases} 2f_c & k = 0 \\ 2f_c \dfrac{\sin(2\pi f_c k)}{2\pi f_c k} & k \neq 0 \end{cases}$$

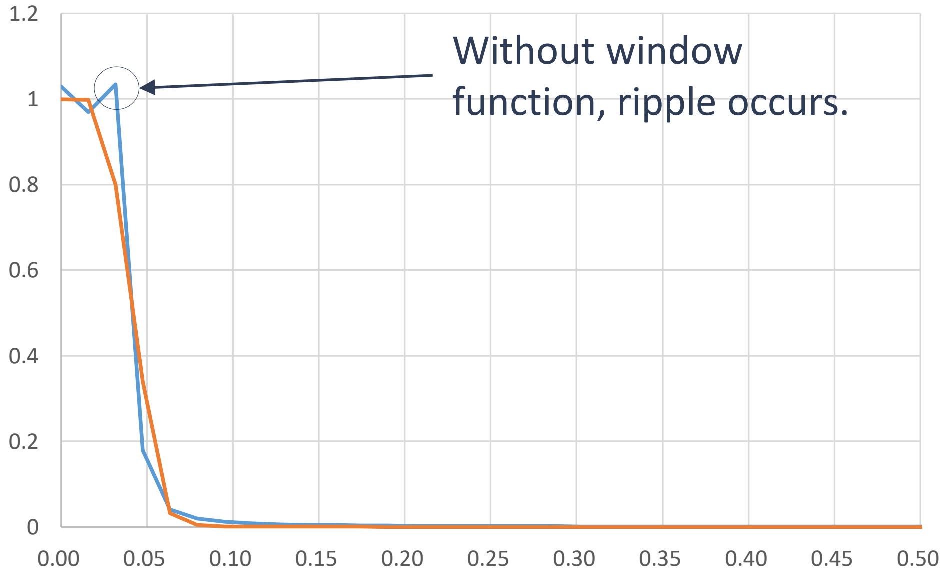
$-\frac{M}{2} \leq k \leq \frac{M}{2}$

$k\ is\ integer$

$$h_m = w_m h_m$$

$$w_m = 0.5 - 0.5cos\left(\frac{2\pi m}{M}\right) \quad m = 0 \ldots M$$





FIR High Pass Filter

Example

F_c=1000 : Cutoff Frequency

F_s=48000 : Sampling Frequency

$f_c = \frac{F_c}{F_s}$ =0.020833 : Normalized Cutoff Frequency

M=63 : Taps

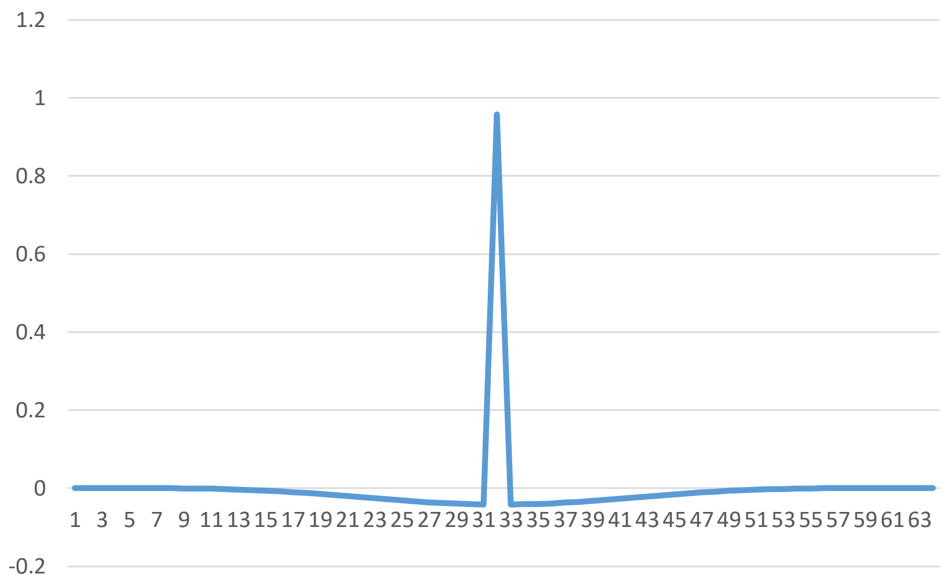
$$h_{k+M/2} = \begin{cases} 1 - 2f_c & k = 0 \\ \dfrac{\sin(\pi k)}{\pi k} - 2f_c \dfrac{\sin(2\pi f_c k)}{2\pi f_c k} & k \neq 0 \end{cases}$$

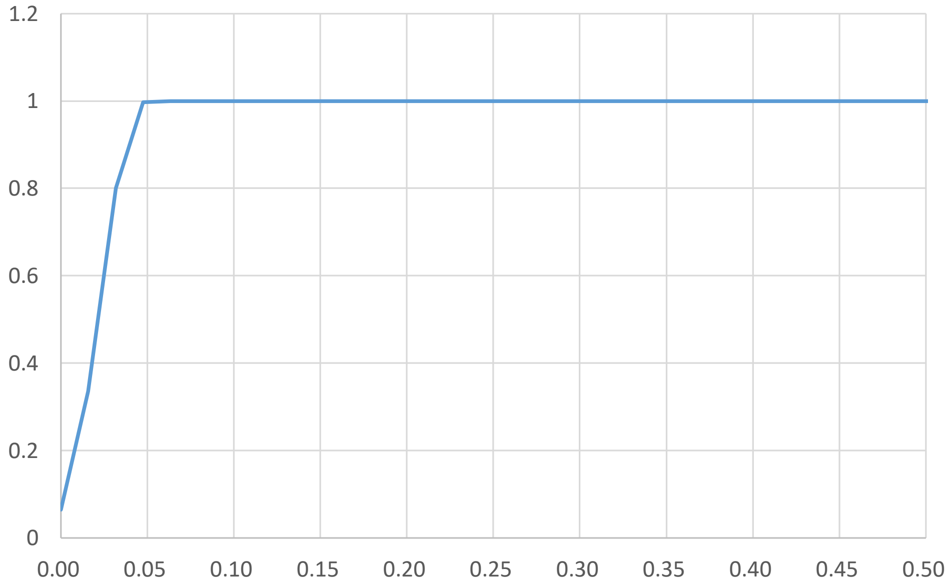
$-\frac{M}{2} \leq k \leq \frac{M}{2}$

k is integer

$$h_m = w_m h_m$$

$$w_m = 0.5 - 0.5cos\left(\frac{2\pi m}{M}\right) \quad m = 0 \ldots M$$





FIR Band Pass Filter

Example

F_L=1000 : Low side Cutoff Frequency

F_H=2000 : High side Cutoff Frequency

F_s=48000 : Sampling Frequency

$f_L = \frac{F_L}{F_s}$ =0.020833 : Low side Normalized Cutoff Freq.

$f_H = \frac{F_H}{F_s}$ =0. 041667 : High side Normalized Cutoff Freq.

M=63 : Taps

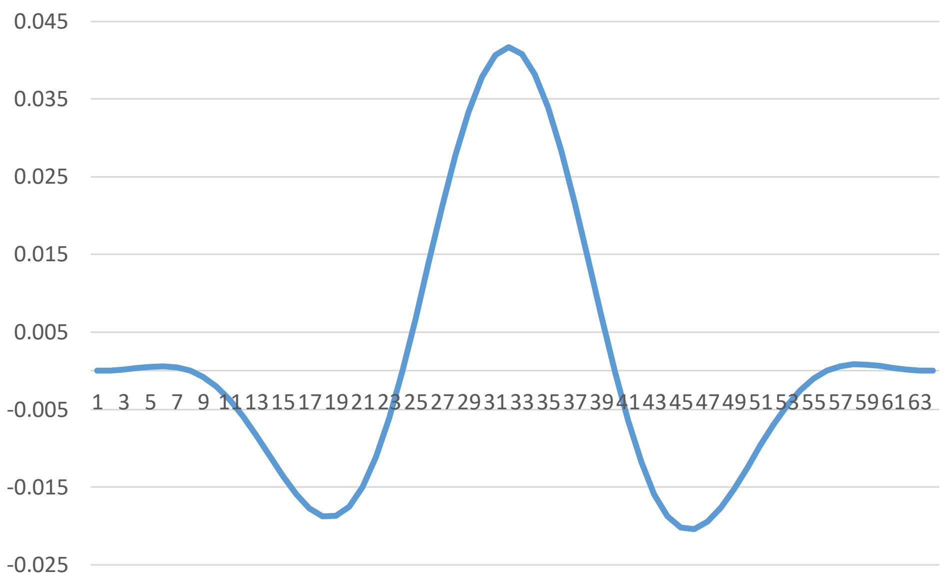
$$h_{k+M/2} = \begin{cases} 2(f_H - f_L) & k = 0 \\ 2f_h \dfrac{\sin(2\pi f_h k)}{2\pi f_h k} - 2f_l \dfrac{\sin(2\pi f_l k)}{2\pi f_l k} & k \neq 0 \end{cases}$$

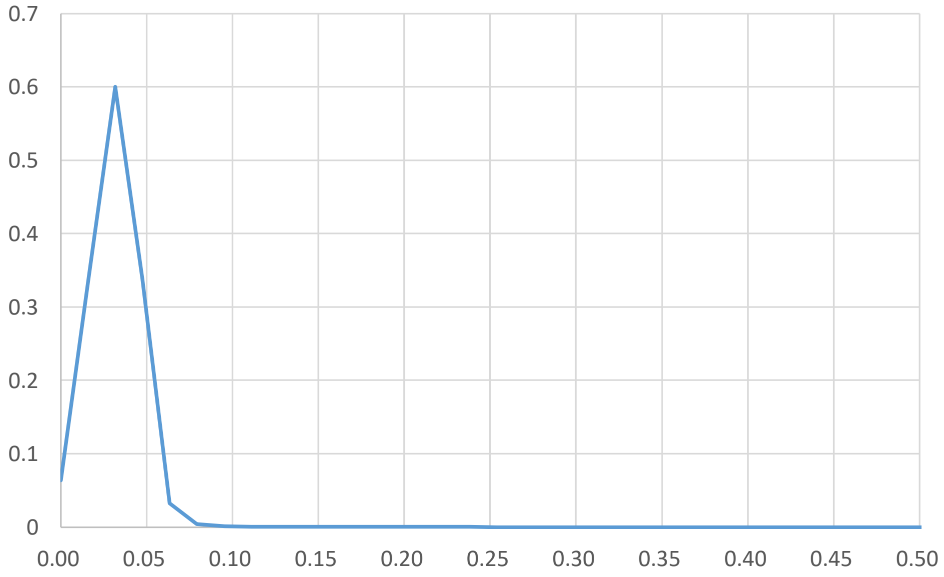
$-\frac{M}{2} \leq k \leq \frac{M}{2}$

$k \ in \ integer$

$$h_m = w_m h_m$$

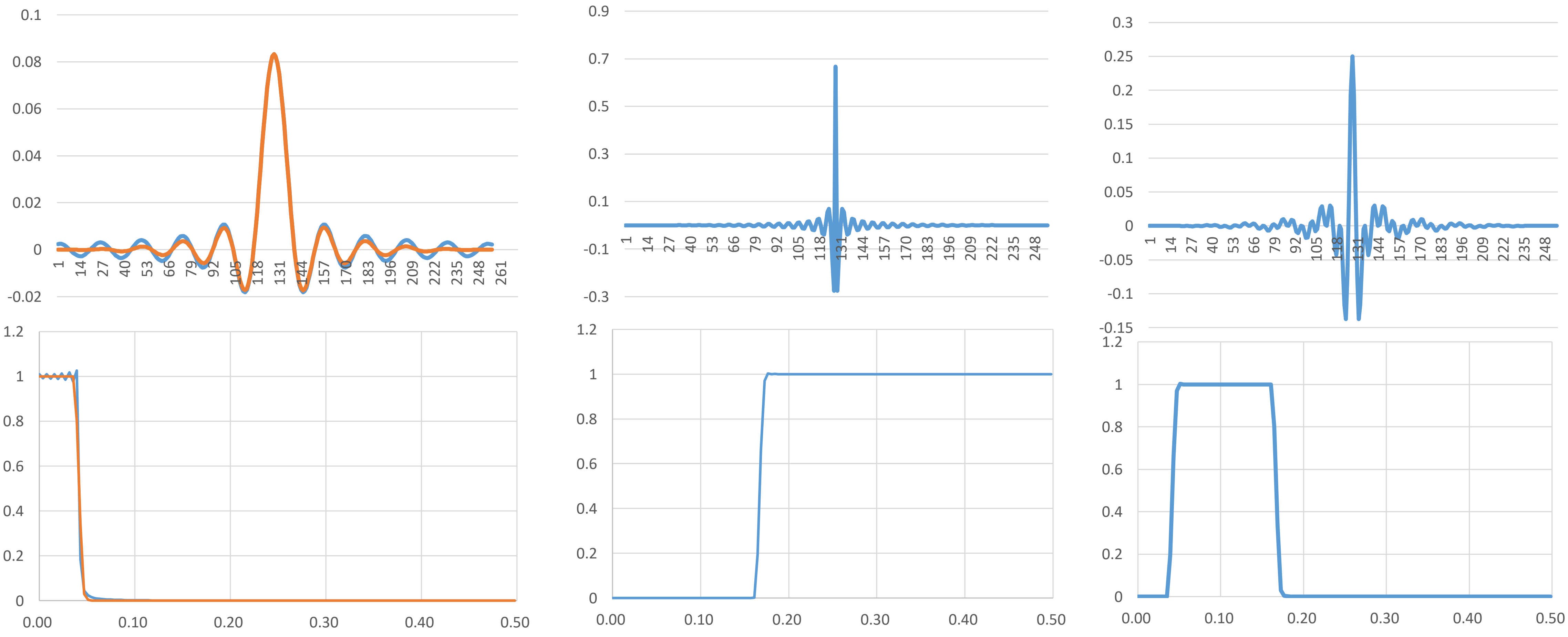
$$w_m = 0.5 - 0.5cos\left(\frac{2\pi m}{M}\right) \quad m = 0 \ldots M$$





FIR Digital Filter

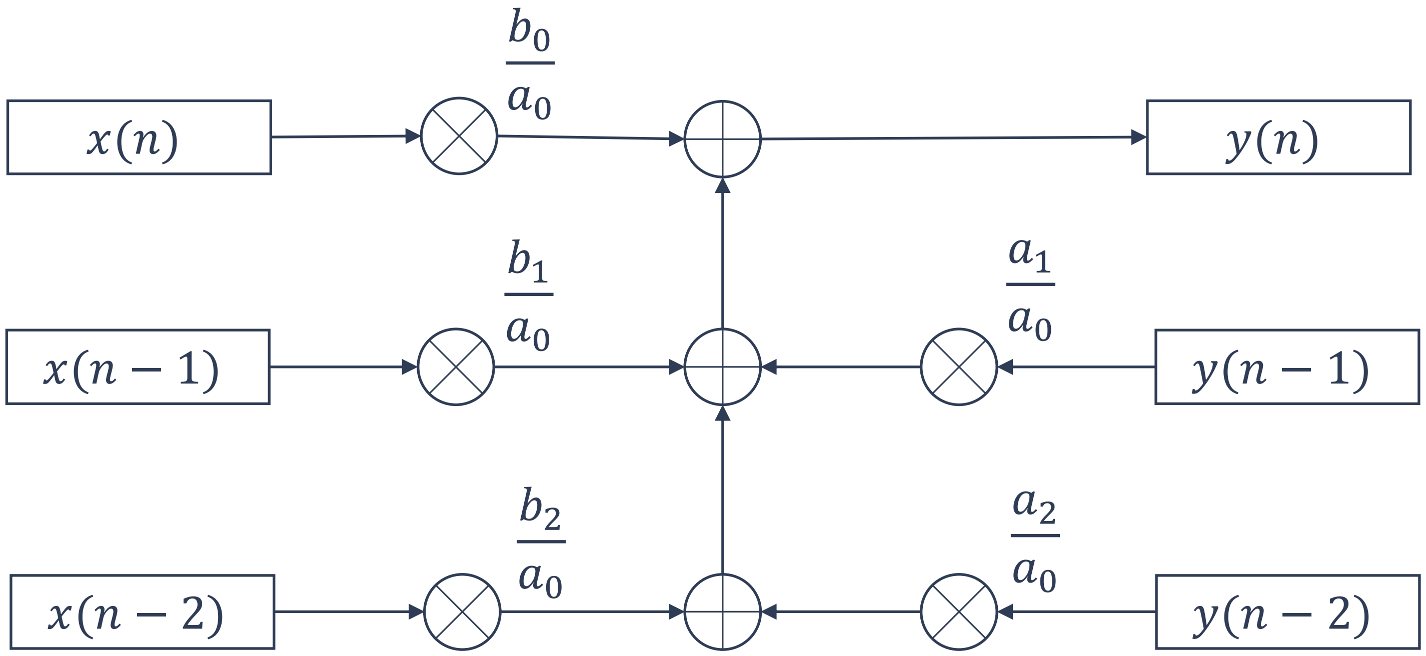
Increasing FIR filter taps improves the characteristics. The graphs show the characteristics of LPF, HPF and BPF in 255 taps



IIR Digital Filter

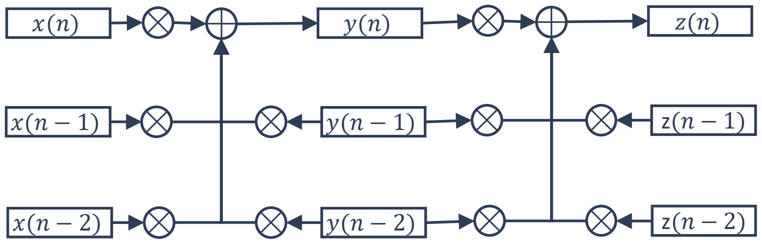
Modified IIR Filter: Biquad Filter

$$y(n) = \frac{b_0}{a_0}x(n) + \frac{b_1}{a_0}x(n-1) + \frac{b_2}{a_0}x(n-2) - \frac{a_1}{a_0}y(n-1) - \frac{a_2}{a_0}y(n-2)$$



In addition to requiring fewer computational resources, they are widely used because they can be cascaded. Cascading can also yield steeper filter characteristics.

Cascaded Biquad Filter



IIR (Biquand) Low Pass Filter

$$y(n) = \frac{b_0}{a_0}x(n) + \frac{b_1}{a_0}x(n-1) + \frac{b_2}{a_0}x(n-2) - \frac{a_1}{a_0}y(n-1) - \frac{a_2}{a_0}y(n-2)$$

Example

F_c=4000 : Cutoff Frequency

F_s=48000 : Sampling Frequency

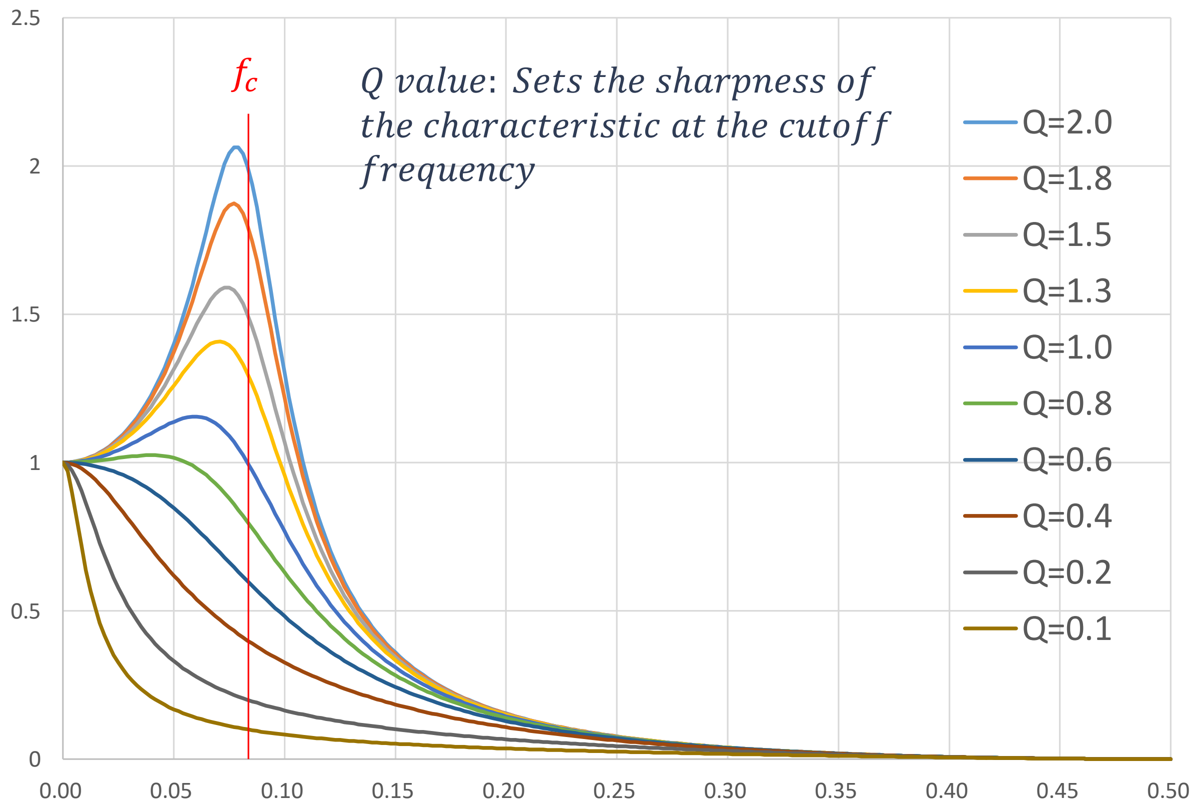
$f_c = \frac{F_c}{F_s}$ =0.08333 : Normalized Cutoff Frequency

$$\omega_c = 2\pi\frac{f_c}{f_s} \qquad alpha = \frac{\sin(w_c)}{2Q}$$

$$b_0 = \frac{1-\cos(\omega_c)}{2} \qquad a_0 = 1 + alpha$$

$$b_1 = 1 - \cos(\omega_c) \qquad a_1 = -2\cos(\omega_c)$$

$$b_2 = \frac{1-\cos(\omega_c)}{2} \qquad a_2 = 1 - alpha$$



IIR (Biquand) High Pass Filter

Example

$$y(n) = \frac{b_0}{a_0}x(n) + \frac{b_1}{a_0}x(n-1) + \frac{b_2}{a_0}x(n-2) - \frac{a_1}{a_0}y(n-1) - \frac{a_2}{a_0}y(n-2)$$

F_c=1000 : Cutoff Frequency

F_s=48000 : Sampling Frequency

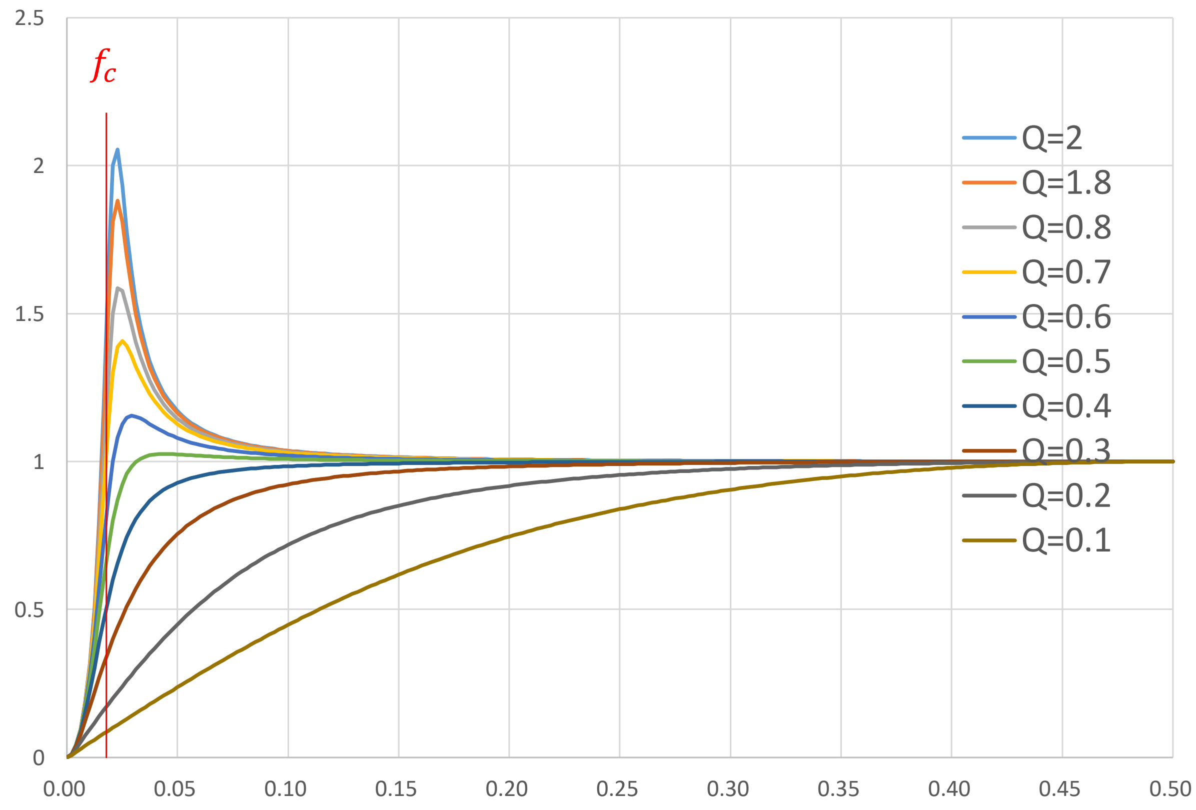
$f_c = \frac{F_c}{F_s}$=0.020833 : Normalized Cutoff Frequency

$$\omega_c = 2\pi\frac{f_c}{f_s} \qquad alpha = \frac{\sin(w_c)}{2Q}$$

$$b_0 = \frac{1+\cos(\omega_c)}{2} \qquad a_0 = 1 + alpha$$

$$b_1 = -(1+\cos(\omega_c)) \qquad a_1 = -2\cos(\omega_c)$$

$$b_2 = \frac{1+\cos(\omega_c)}{2} \qquad a_2 = 1 - alpha$$



IIR (Biquand) Band Pass Filter

Example

$$y(n) = \frac{b_0}{a_0}x(n) + \frac{b_1}{a_0}x(n-1) + \frac{b_2}{a_0}x(n-2) - \frac{a_1}{a_0}y(n-1) - \frac{a_2}{a_0}y(n-2)$$

$$\omega_c = 2\pi\frac{f_c}{f_s}$$

$$alpha = \sin(\omega_c)\sinh\left(\frac{\ln(2)}{2} \times Bandwidth \times \frac{\omega_c}{\sin(\omega_c)}\right)$$

$$b_0 = alpha \qquad a_0 = 1 + alpha$$

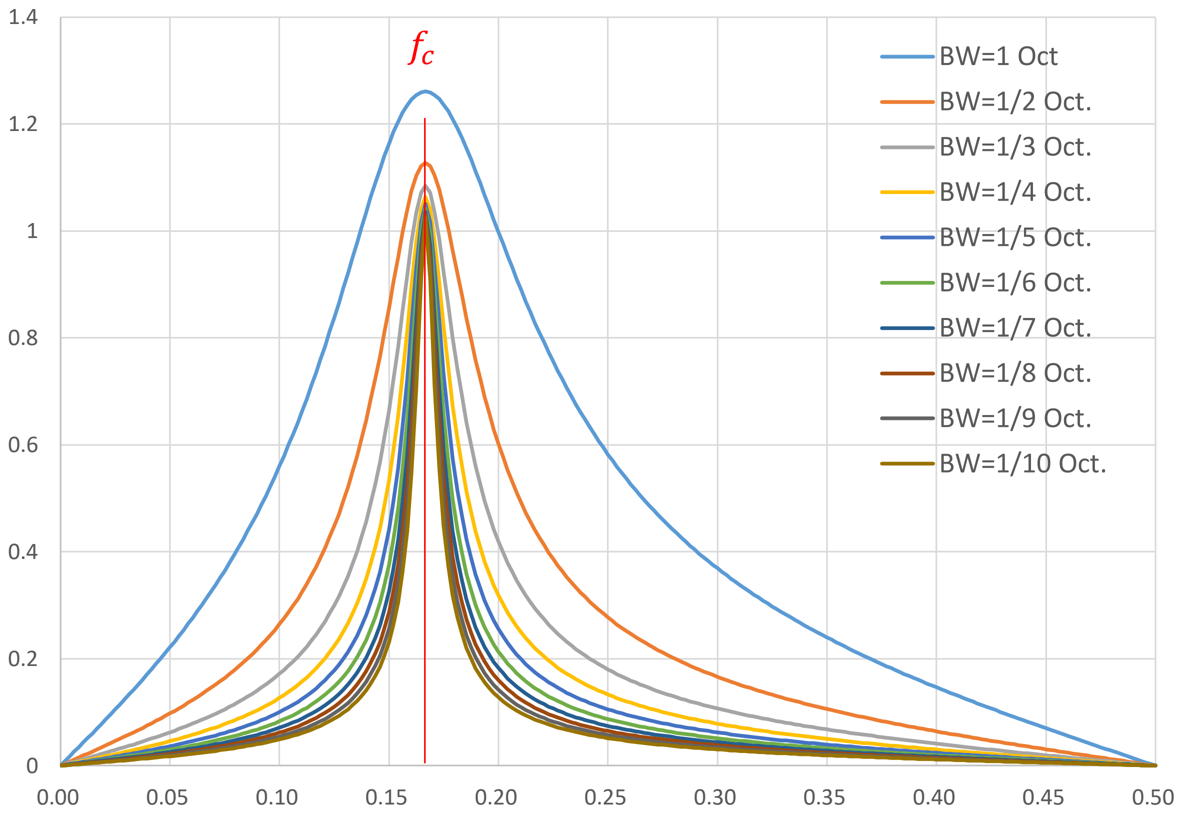
$$b_1 = 0 \qquad a_1 = -2\cos(\omega_c)$$

$$b_2 = -alpha \qquad a_2 = 1 - alpha$$

F_c=8000 : Cutoff Frequency

F_s=48000 : Sampling Frequency

$f_c = \frac{F_c}{F_s}$=0.166667 : Normalized Cutoff Frequency



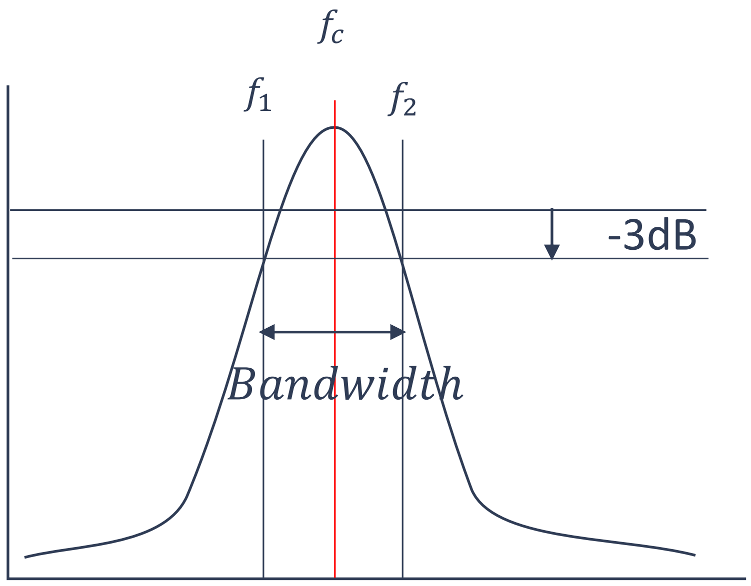
IIR (Biquand) Band Pass Filter

How to set “Bandwidth” for the biquad bandpass filter

$$\omega_c = 2\pi \frac{f_c}{f_s}$$

$$alpha = \sin(\omega_c)\sinh\left(\frac{\ln(2)}{2} \times \boxed{Bandwidth} \times \frac{\omega_c}{\sin(\omega_c)}\right)$$

Bandwidth is specified in octaves



Bandwidth for 1 octave

$$f_2 = 2f_1$$

$$f_c = \sqrt{f_1 f_2} = \sqrt{2} f_1 = \frac{f_2}{\sqrt{2}}$$

$$Bandwidth = f_2 - f_1 = \frac{1}{\sqrt{2}} f_c$$

Bandwidth for $\frac{1}{n}$ octave

$$f_2 = \sqrt[n]{2} f_1$$

$$f_c = \sqrt{f_1 f_2} = \sqrt[2n]{2} f_1 = \frac{f_2}{\sqrt[2n]{2}}$$

$$Bandwidth = f_2 - f_1 = \frac{\sqrt[n]{2} - 1}{\sqrt[2n]{2}} f_c$$

IIR (Biquad) Notch Filter

Example

$$y(n) = \frac{b_0}{a_0}x(n) + \frac{b_1}{a_0}x(n-1) + \frac{b_2}{a_0}x(n-2) - \frac{a_1}{a_0}y(n-1) - \frac{a_2}{a_0}y(n-2)$$

$$\omega_c = 2\pi\frac{f_c}{f_s}$$

$$alpha = \sin(\omega_c)\sinh\left(\frac{\ln(2)}{2} \times Bandwidth \times \frac{\omega_c}{\sin(\omega_c)}\right)$$

$$b_0 = 1 \qquad a_0 = 1 + alpha$$

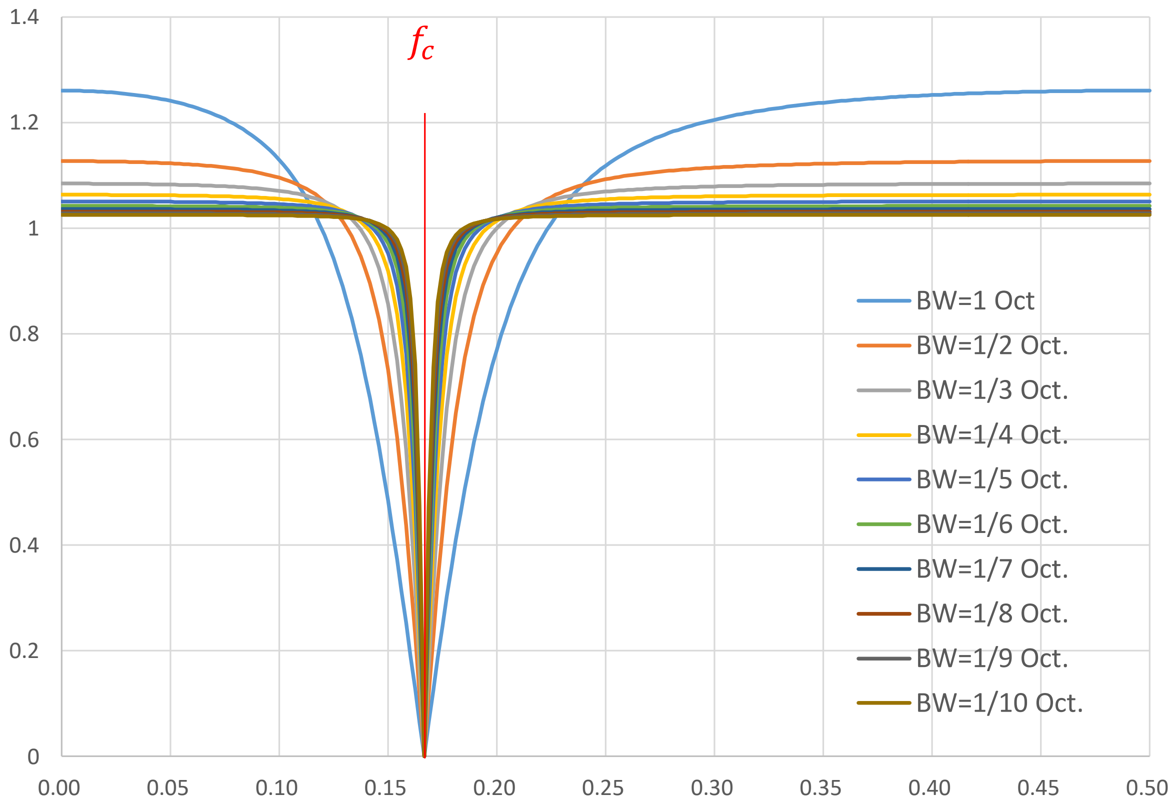
$$b_1 = -2\cos(\omega_c) \qquad a_1 = -2\cos(\omega_c)$$

$$b_2 = 1 \qquad a_2 = 1 - alpha$$

F_c=8000 : Cutoff Frequency

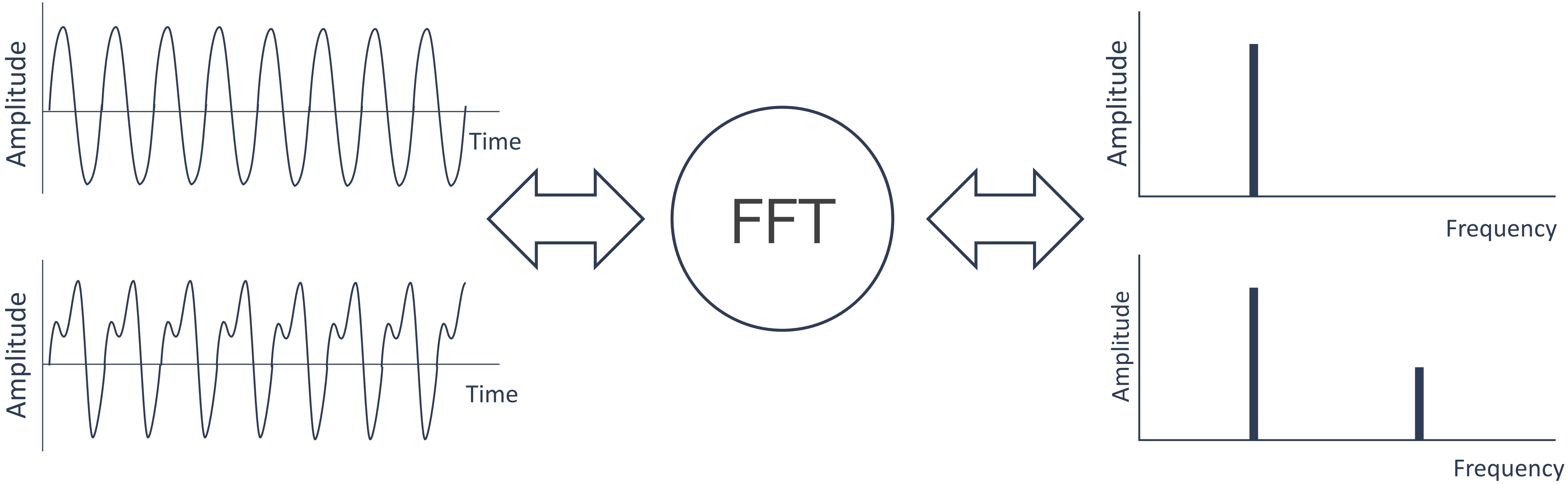
F_s=48000 : Sampling Frequency

$f_c = \frac{F_c}{F_s}$=0.166667 : Normalized Cutoff Frequency



Fast Fourier Transform (FFT)

FFT is an algorithm for high-speed conversion of observed digital data in time-space into frequency space

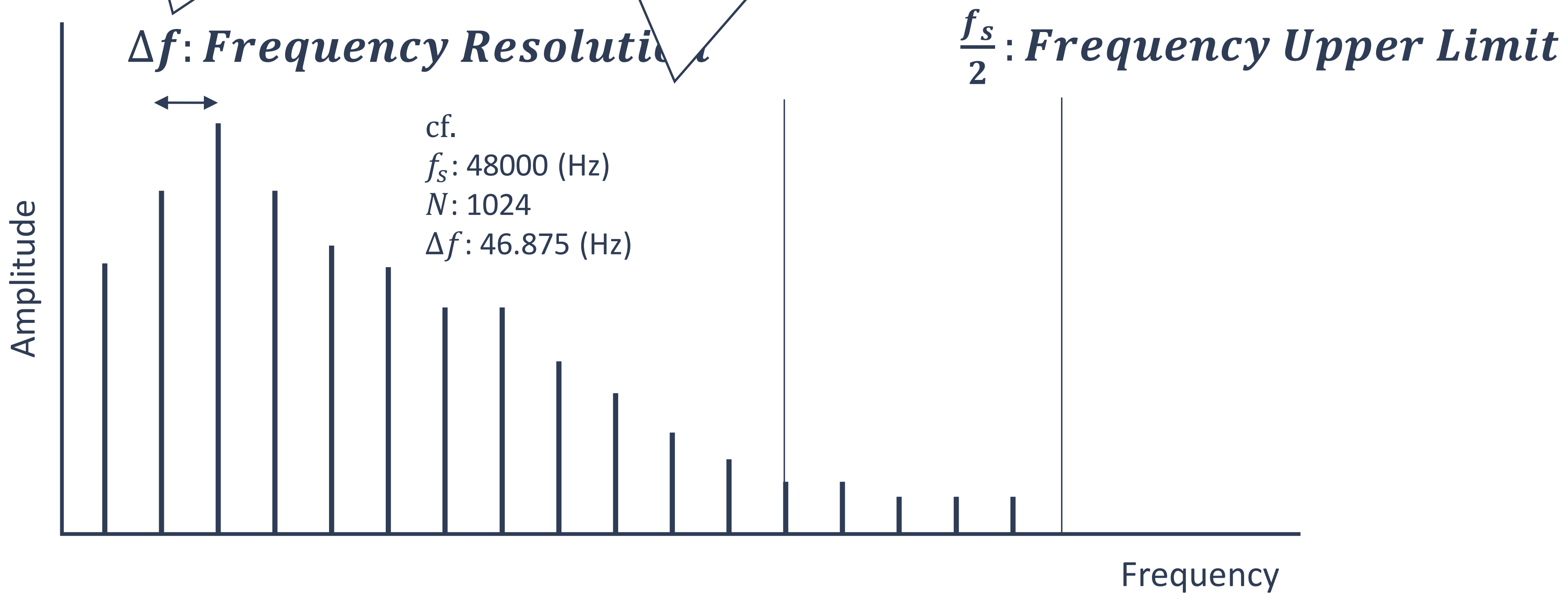


Fast Fourier Transform

In vibration analysis, frequencies above the upper analytical frequency limit are often attenuated by LPF

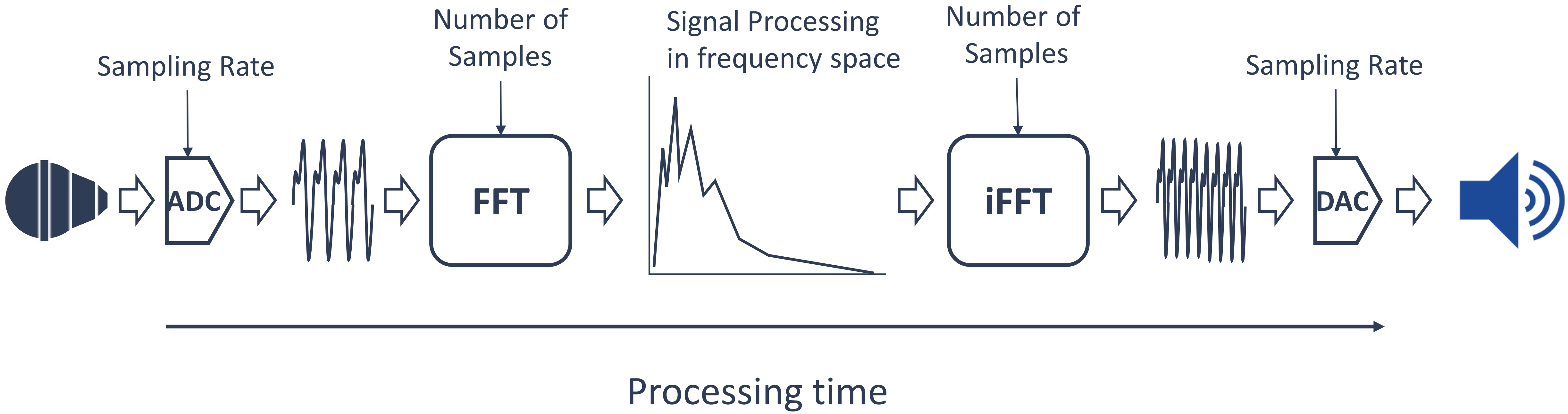
$$\Delta f = \frac{f_s}{N} = \frac{\textit{Samling Frequency}}{\textit{Number of Samples}}$$

$\frac{f_s}{2.56}$: *Analysis Frequency Upper limit*



Short Time Fourier Transform

The sampling rate and number of samples determine whether processing can be done in a time that does not cause perceptible delay.



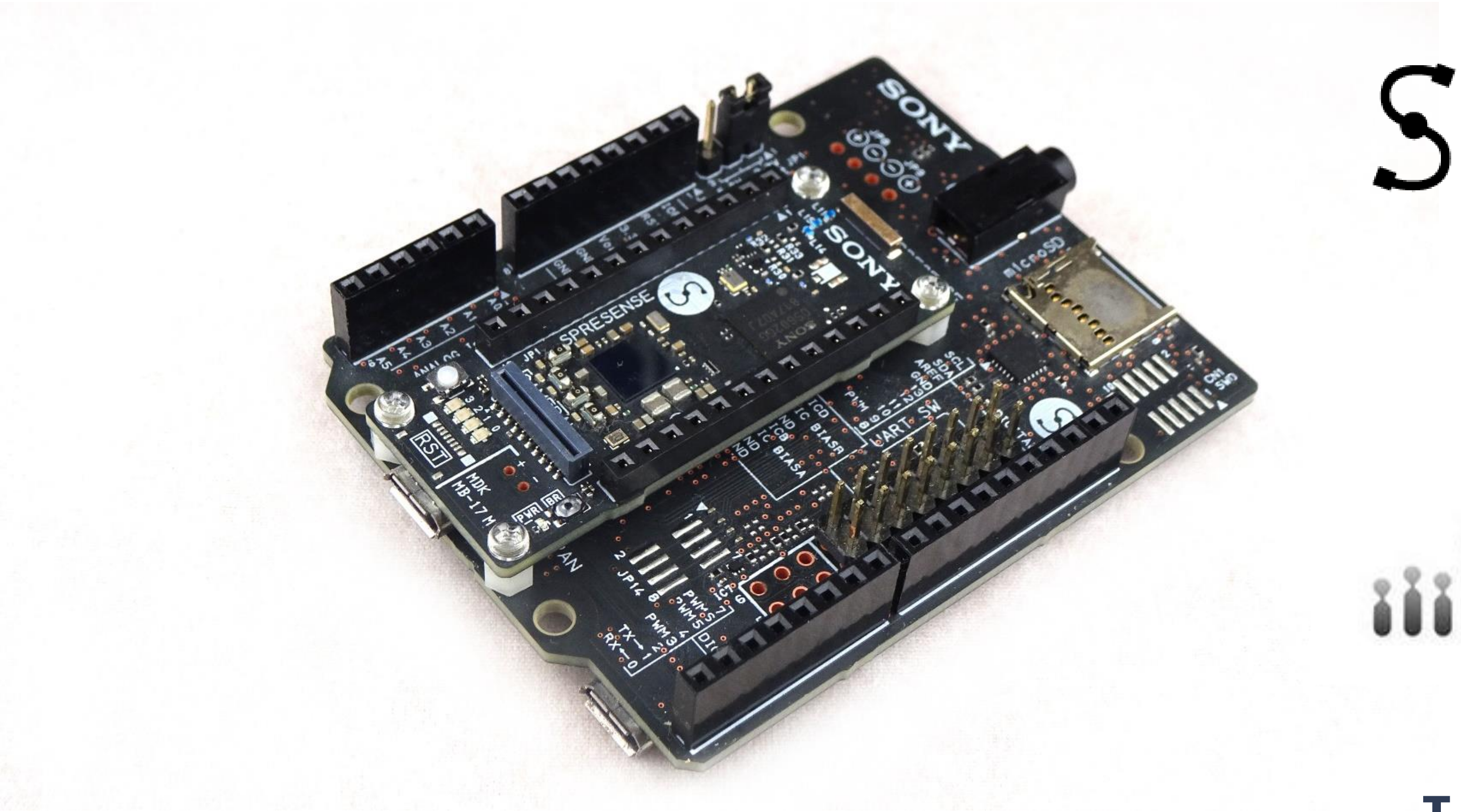
Short Time Fourier Transform

STFT（Short Time Fourier Transform）is an FFT that is performed in a short time (small number of samples). For real-time signal processing, it is used to reduce latency.

From Yamaha's paper, 30msec can be used as one guideline for the amount of delay, since people cannot perceive delay in musical instrument performance if it is within 30msec.

Number of Samples		256	512	1024	2048	4096
Sampling Rate	48000(Hz)	5.3 msec	10.6 msec	21.3 msec	42.7 msec	85.3 msec
	192000(Hz)	1.3 msec	2.7 msec	5.3 msec	10.6 msec	21.3 msec

Latency



SPRESENSE

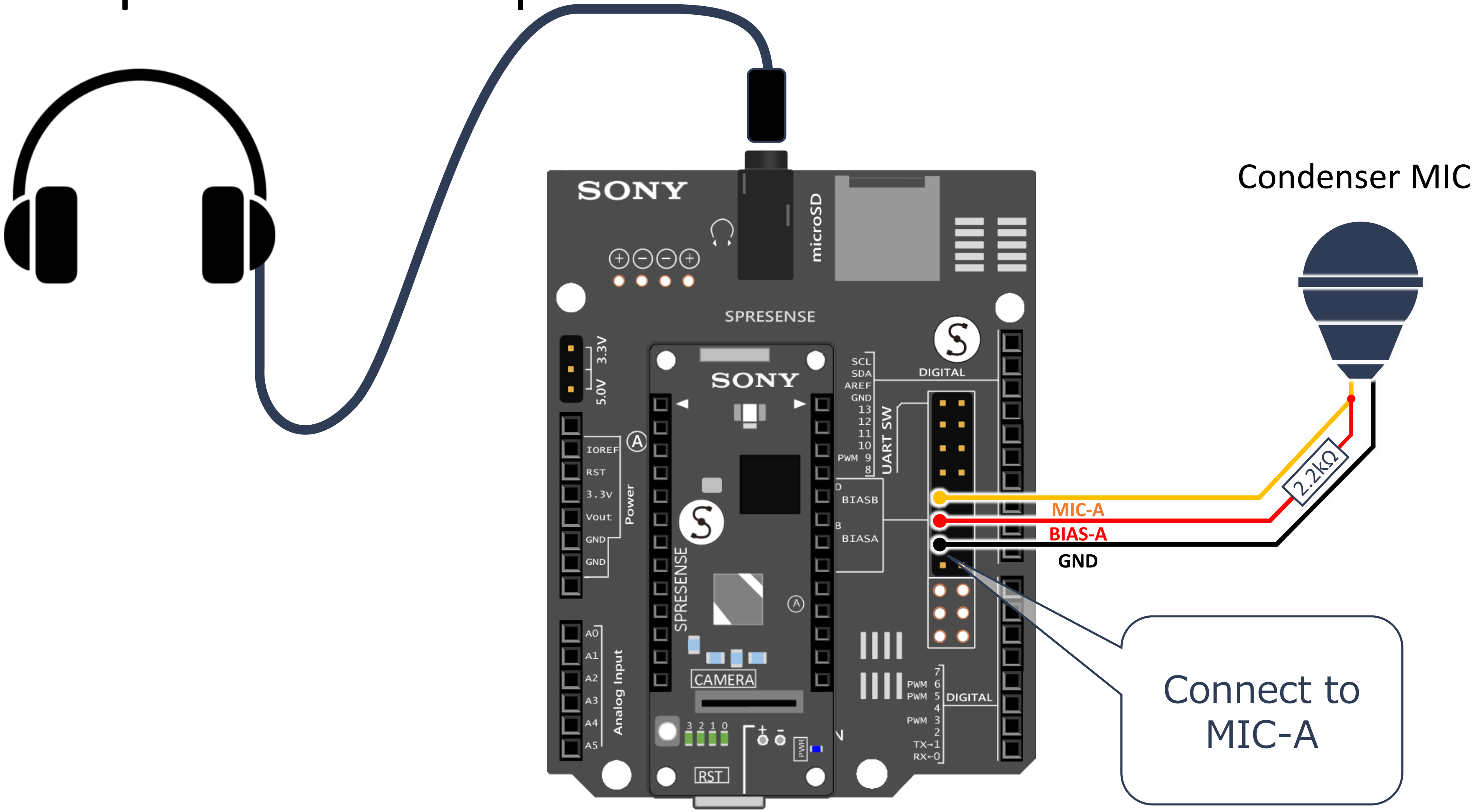
Digital Filter Implementation on Spresense

Low latency input and output by Spresense

"Frontend" library for low-latency input/output with Spresense

Direct connection of the front-end library to the mixer library enables low latency of less than 1 ms, allowing sufficient processing time for digital filters.

Connection of a microphone and a headphone to Spresense



Implementation of Low latency Audio I/O

Spresense_FrontEnd_through.ino -(1)

```
... snip ...
#define SAMPLE_SIZE (720)

FrontEnd *theFrontEnd;
OutputMixer *theMixer;
const int32_t channel_num = AS_CHANNEL_MONO;
const int32_t bit_length  = AS_BITLENGTH_16;
const int32_t sample_size = SAMPLE_SIZE;
const int32_t frame_size  = sample_size * (bit_length / 8) * channel_num;
... snip ...
```

```
static void frontend_pcm_cb(AsPcmDataParam pcm) {
  static uint8_t mono_input[frame_size];
  static uint8_t stereo_output[frame_size*2];

  frontend_signal_input(pcm, mono_input, frame_size);
  signal_process((int16_t*)mono_input, (int16_t*)stereo_output, sample_size);
  mixer_stereo_output(stereo_output, frame_size);
  return;
}
```

Function called when a set number of samples of data has been obtained. Digital filter processing is performed in this function.

```
void frontend_signal_input(AsPcmDataParam pcm, uint8_t* input, uint32_t frame_size) {
  memset(input, 0, frame_size);
  if (pcm.size != 0)
    memcpy(input, pcm.mh.getPa(), pcm.size); // copy the signal to signal_input buffer
  return;
}
```

Function to copy the retrieved samples to a buffer

```
void signal_process(int16_t* mono_input, int16_t* stereo_output, uint32_t sample_size) {
  // TODO: add digital filters
  // copy the signal to output stereo buffer
  for (int n = SAMPLE_SIZE-1; n >= 0; --n)  {
    stereo_output[n*2] = stereo_output[n*2+1] = mono_input[n];  // audio through
  }
  return;
}
```

Copy input data to buffer for output
Converts monaural data to stereo

```
void mixer_stereo_output(uint8_t* stereo_output, uint32_t frame_size) {
  AsPcmDataParam pcm_param;
  if (pcm_param.mh.allocSeg(S0_REND_PCM_BUF_POOL, frame_size) != ERR_OK)   return;

  pcm_param.is_end = false;
  pcm_param.identifier = OutputMixer0;
  pcm_param.callback = 0;
  pcm_param.bit_length = bit_length;
  pcm_param.size = frame_size*2;
  pcm_param.sample = frame_size;
  pcm_param.is_valid = true;

  memcpy(pcm_param.mh.getPa(), stereo_output, pcm_param.size);
  theMixer->sendData(OutputMixer0, outputmixer0_send_cb, pcm_param);
  return;
}
```

Output data set in buffer to headphone output

Implementation of Low latency Audio I/O

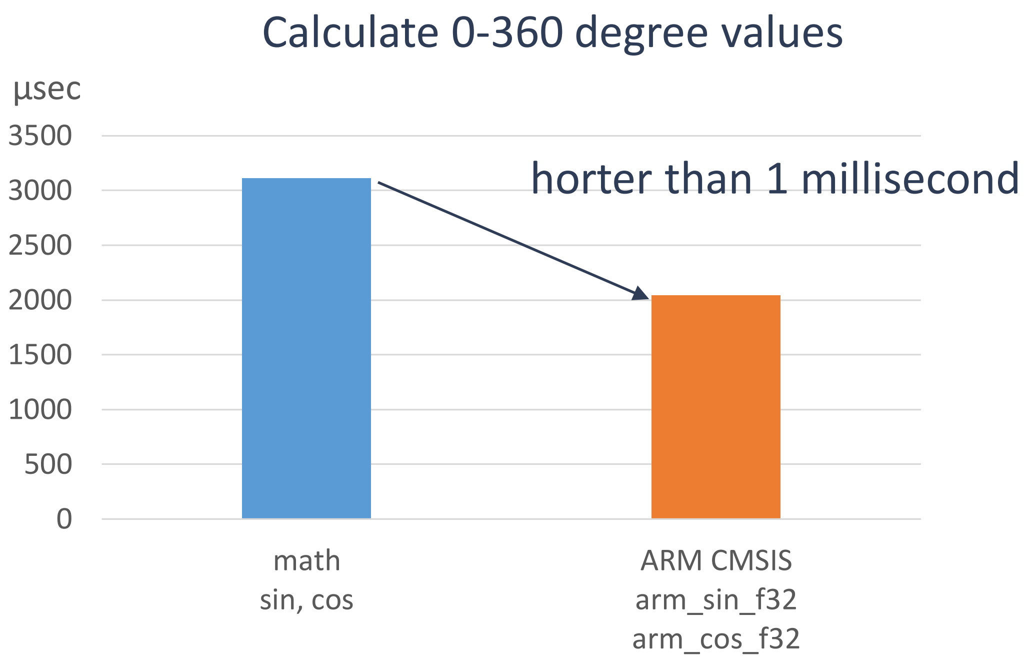
Spresense_FrontEnd_through.ino -(2)

```
void setup() {
  initMemoryPools();
  createStaticPools(MEM_LAYOUT_RECORDINGPLAYER);

  theFrontEnd = FrontEnd::getInstance();
  theMixer = OutputMixer::getInstance();

  theFrontEnd->setCapturingClkMode(FRONTEND_CAPCLK_NORMAL);
  theFrontEnd->begin(frontend_attention_cb);
  theMixer->begin();

  theFrontEnd->setMicGain(0);
  theFrontEnd->activate(frontend_done_cb, true);
  theMixer->create(mixer_attention_cb);
  theMixer->activate(OutputMixer0, outputmixer_done_cb);
  delay(100); /* waiting for Mic startup */

  AsDataDest dst;
  dst.cb = frontend_pcm_cb;
  theFrontEnd->init(channel_num, bit_length, sample_size, AsDataPathCallback, dst);

  theMixer->setVolume(-10, -10, -10); /* -10dB */
  board_external_amp_mute_control(false);
  theFrontEnd->start();                              FrontEnd and Mixer settings
}

void loop() {}
```

ARM CMSIS DSP Library

ARM CMSIS DSP is a library for fast numerical operations

```
#include <math.h>
#define ARM_MATH_CM4
#define __FPU_PRESENT 1U
#include <cmsis/arm_math.h>

void setup() {
  Serial.begin(115200);
  uint32_t start_time, duration;
  start_time = micros();
  for (int x = 0; x < 360; ++x) {
    float radian = x * M_PI/180.0;
    float sin_y = sin(radian);
    float cos_y = cos(radian);                    GCC Math Library
  }
  duration = micros() - start_time;
  Serial.println("math duration: " + String(duration));

  start_time = micros();
  for (int x = 0; x < 360; ++x) {
    float radian = x * M_PI/180.0;
    float sin_y = arm_sin_f32(radian);
    float cos_y = arm_cos_f32(radian);            ARM CMSIS DSP API
  }
  duration = micros() - start_time;
  Serial.println("arm_math duration: " + String(duration));
}

void loop() {  }
```



FIR filter implementation with ARM CMSIS

```
void arm_fir_init_f32 (
    arm_fir_instance_f32 *    S,
    uint16_t                  numTaps,
    const float32_t *         pCoeffs,
    float32_t *               pState,
    uint32_t                  blockSize
)
```

[in,out] S	points to an instance of the floating-point FIR filter structure
[in] numTaps	number of filter coefficients in the filter
[in] pCoeffs	points to the filter coefficients buffer
[in] pState	points to the state buffer
[in] blockSize	number of samples processed per call

Details

pCoeffs points to the array of filter coefficients stored in time reversed order: {b[numTaps-1], b[numTaps-2], b[N-2], ..., b[1], b[0]} pState points to the array of state variables. pState is of length numTaps+blockSize-1 samples, where blockSize is the number of input samples processed by each call to arm_fir_f32().

```
void arm_fir_f32  (
   const arm_fir_instance_f32 *    S,
   const float32_t *               pSrc,
   float32_t *                     pDst,
   uint32_t                        blockSize
)
```

[in]	S	points to an instance of the floating-point FIR filter structure
[in]	pSrc	points to the block of input data
[out]	pDst	points to the block of output data
[in]	blockSize	number of samples to process

Remarks

A faster function, arm_fir_fast_q15(), can also be used, but is less accurate.

FIR Low Pass Filter implementation

Example: Spresense_FrontEnd_FIR_LPF.ino

Initialization of FIR filter

```
#define TAPS 63
arm_fir_instance_f32 S;
float pCoeffs[TAPS];
float pState[TAPS+SAMPLE_SIZE];
…
void initializeFirLPF() {
  const uint32_t CUTTOFF_FREQ_HZ = 1000;
  float Fc = (float)CUTTOFF_FREQ_HZ/AS_SAMPLINGRATE_48000;
  const int H_TAPS = TAPS/2;

  int n = 0;
  for (int k = H_TAPS; k >= -H_TAPS; --k) {
    if (k == 0) pCoeffs[k] = 2.*Fc;
    else   pCoeffs[n++] = 2.*Fc*arm_sin_f32(2.*PI*Fc*k)/2*PI*Fc*k;
  }
                                          coefficient calculation

  for (int m = 0; m < TAPS; ++m) {
    pCoeffs[m] = (0.5 - 0.5*arm_cos_f32(2*PI*m/TAPS))*pCoeffs[m];
  }
                                          Window function multiplication

  arm_fir_init_f32(&S, TAPS, pCoeffs, pState, SAMPLE_SIZE);   structure initialization
}
```

Implementation of the “signal_processing” function

```
void signal_process(int16_t* mono_input, int16_t* stereo_output, uint32_t sample_size) {

  static float pSrc[SAMPLE_SIZE];
  static float pDst[SAMPLE_SIZE];
  q15_t* q15_mono = (q15_t*)mono_input;
  arm_q15_to_float(&q15_mono[0], &pSrc[0], SAMPLE_SIZE);
  arm_fir_f32(&S, &pSrc[0], &pDst[0], SAMPLE_SIZE);
  arm_float_to_q15(&pSrc[0], &q15_mono[0], SAMPLE_SIZE);
  mono_input = (int16_t*)q15_mono;        Applying the FIR Low Pass Filter

  /* clean up the output buffer */
  memset(stereo_output, 0, sizeof(int16_t)*sample_size*2);

  /* copy the signal to output buffer */
  for (int n = SAMPLE_SIZE-1; n >= 0; --n)  {
    stereo_output[n*2] = stereo_output[n*2+1] = mono_input[n];
  }

  return;
}
```

FIR High Pass Filter implementation

Example: Spresense_FrontEnd_FIR_HPF.ino

Initialization of FIR filter

```
#define TAPS 63
arm_fir_instance_f32 S;
float pCoeffs[TAPS];
float pState[TAPS+SAMPLE_SIZE];
…
void initializeFirHPF() {
  const uint32_t CUTTOFF_FREQ_HZ = 1000;
  float Fc = (float)CUTTOFF_FREQ_HZ/AS_SAMPLINGRATE_48000;
  const int H_TAPS = TAPS/2;

  int n = 0;
  for (int k =-H_TAPS; k >= -H_TAPS; --k) {
    if (k == 0) pCoeffs[k] = 1. - 2.*Fc;
    else   pCoeffs[n++] = arm_sin_f32(PI*k)/PI*k  - 2.*Fc*arm_sin_f32(2.*PI*Fc*k)/2*PI*Fc*k;
  }
                                                              coefficient calculation

  for (int m = 0; m < TAPS; ++m) {
    pCoeffs[m] = (0.5 - 0.5*arm_cos_f32(2*PI*m/TAPS))*pCoeffs[m];
  }

  arm_fir_init_f32(&S, TAPS, pCoeffs, pState, SAMPLE_SIZE);
}
```

Implementation of the “signal_processing” function

```
void signal_process(int16_t* mono_input, int16_t* stereo_output, uint32_t sample_size) {

  static float pSrc[SAMPLE_SIZE];
  static float pDst[SAMPLE_SIZE];
  q15_t* q15_mono = (q15_t*)mono_input;
  arm_q15_to_float(&q15_mono[0], &pSrc[0], SAMPLE_SIZE);
  arm_fir_f32(&S, &pSrc[0], &pDst[0], SAMPLE_SIZE);
  arm_float_to_q15(&pSrc[0], &q15_mono[0], SAMPLE_SIZE);
  mono_input = (int16_t*)q15_mono;


  /* clean up the output buffer */
  memset(stereo_output, 0, sizeof(int16_t)*sample_size*2);

  /* copy the signal to output buffer */
  for (int n = SAMPLE_SIZE-1; n >= 0; --n)  {
    stereo_output[n*2] = stereo_output[n*2+1] = mono_input[n];
  }

  return;
}
```

Same as LPF implementation

FIR Band Pass Filter implementation

Example: Spresense_FrontEnd_FIR_BPF.ino

Initialization of FIR filter

```
#define TAPS 63
arm_fir_instance_f32 S;
float pCoeffs[TAPS];
float pState[TAPS+SAMPLE_SIZE];
...
void initializeFirBPF() {
  const uint32_t CUTTOFF_LOW_FREQ_HZ = 1000;
  const uint32_t CUTTOFF_HIGH_FREQ_HZ = 2000;
  float Fl = (float)CUTTOFF_LOW_FREQ_HZ/AS_SAMPLINGRATE_48000;
  float Fh = (float)CUTTOFF_HIGH_FREQ_HZ/AS_SAMPLINGRATE_48000;
  const int H_TAPS = TAPS/2;

  int n = 0;
  for (int k = H_TAPS; k >= -H_TAPS; --k) {
    if (k == 0) pCoeffs[n] = 2.*(Fh - Fl);
    else    pCoeffs[n] = 2.*Fh*arm_sin_f32(2.*PI*Fh*k)/(2.*PI*Fh*k)
                       - 2.*Fl*arm_sin_f32(2.*PI*Fl*k)/(2.*PI*Fl*k);
    ++n;
  }
```

coefficient calculation

```
  for (int m = 0; m < TAPS; ++m) {
    pCoeffs[m] = (0.5 - 0.5*arm_cos_f32(2*PI*m/TAPS))*pCoeffs[m];
  }

  arm_fir_init_f32(&S, TAPS, pCoeffs, pState, SAMPLE_SIZE);
}
```

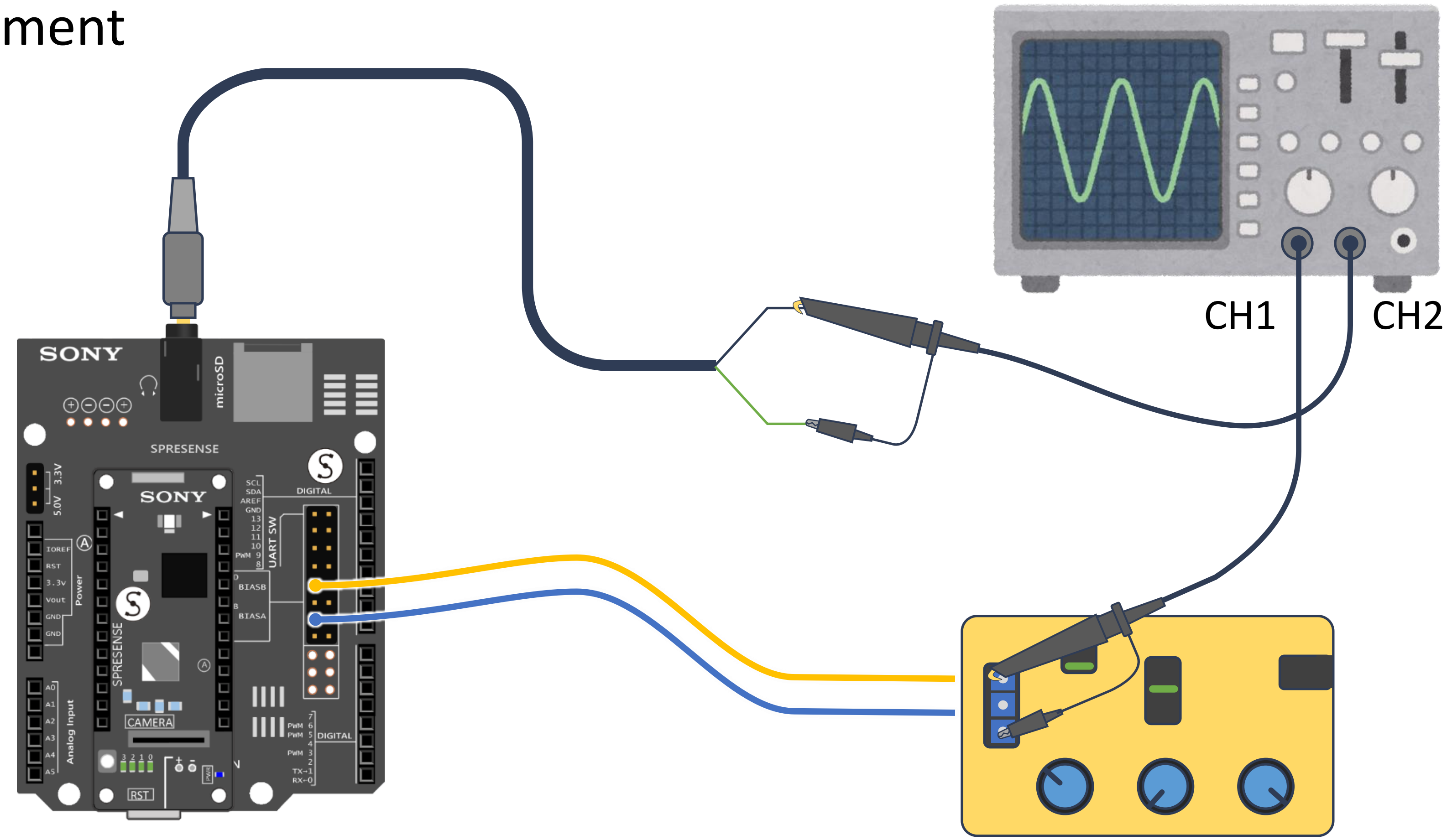
Implementation of the “signal_processing” function

```
void signal_process(int16_t* mono_input, int16_t* stereo_output, uint32_t sample_size) {

  static float pSrc[SAMPLE_SIZE];
  static float pDst[SAMPLE_SIZE];
  q15_t* q15_mono = (q15_t*)mono_input;
  arm_q15_to_float(&q15_mono[0], &pSrc[0], SAMPLE_SIZE);
  arm_fir_f32(&S, &pSrc[0], &pDst[0], SAMPLE_SIZE);
  arm_float_to_q15(&pSrc[0], &q15_mono[0], SAMPLE_SIZE);
  mono_input = (int16_t*)q15_mono;


  /* clean up the output buffer */
  memset(stereo_output, 0, sizeof(int16_t)*sample_size*2);

  /* copy the signal to output buffer */
  for (int n = SAMPLE_SIZE-1; n >= 0; --n)  {
    stereo_output[n*2] = stereo_output[n*2+1] = mono_input[n];
  }

  return;
}
```

Same as LPF implementation

Operation Test for FIR Filter on Spresense

Test Environment



Signal Generator

Operation Test for FIR Filter on Spresense

FIR LPF

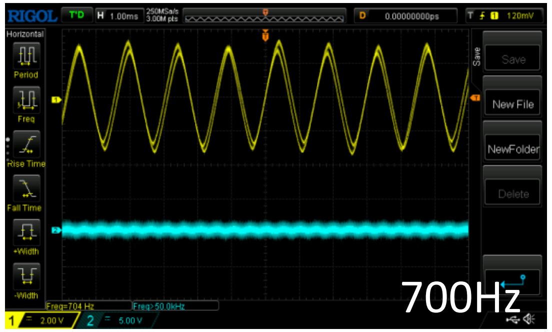
$f_s: 48000Hz$ $f_c: 2000Hz$

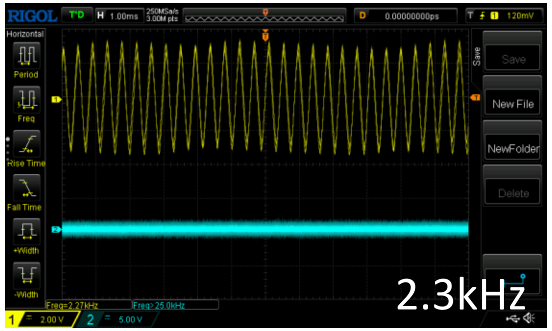
FIR HPF

$f_s: 48000Hz$ $f_c: 1000Hz$

FIR BPF

$f_s: 48000Hz$ $f_l: 1000Hz$ $f_h: 2000Hz$





Biquad IIR filter implementation with ARM CMSIS

```
void arm_biquad_cascade_df2T_init_f32 (
  arm_biquad_cascade_df2T_instance_f32 *   S,
  uint8_t                           numStages,
  const float32_t *                   pCoeffs,
  float32_t *                          pState
)
```

[in,out] S	points to an instance of the filter data structure
[in] numStages	number of 2nd order stages in the filter
[in] pCoeffs	points to the filter coefficients
[in] pState	points to the state buffer

Details

The coefficients are stored in the array pCoeffs in the following order in the not Neon version.

{b10、b11、b12、a11、a12、b20、b21、b22、a21、a22、...}

cf. $y(n) = b10x(n) + b11x(n-1) + b12x(n-2) - a11y(n-1) - a12y(n-2)$

where b1x and a1x are the coefficients for the first stage, b2x and a2x are the coefficients for the second stage, and so on. The pCoeffs array contains a total of 5*numStages values.

```
void arm_biquad_cascade_df2T_f32  (
  const arm_biquad_cascade_df2T_instance_f32 *  S,
  const float32_t *                         pSrc,
  float32_t *                               pDst,
  uint32_t                             blockSize
)
```

[in]	S	points to an instance of the filter data structure
[in]	pSrc	points to the block of input data
[out]	pDst	points to the block of output data
[in]	blockSize	number of samples to process

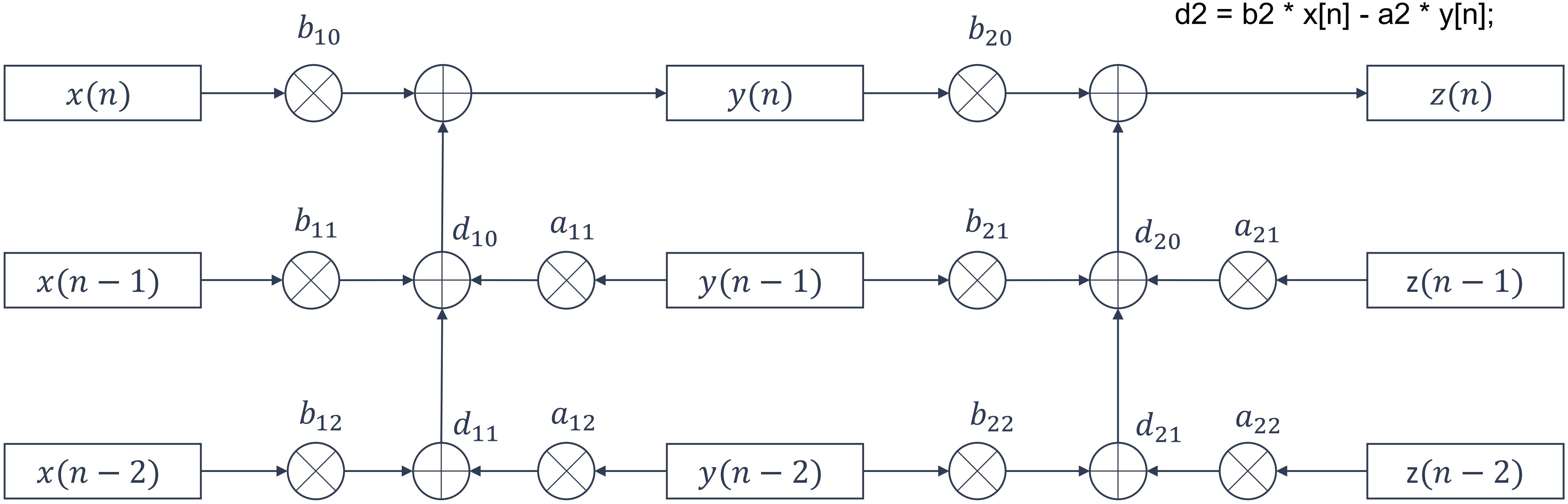
Biquad IIR filter implementation with ARM CMSIS

Coefficients for ARM CMSIS of arm_biquad_cascade_df2T_init_f32

y[n] = b0 * x[n] + d1;

d1 = b1 * x[n] - a1 * y[n] + d2;

d2 = b2 * x[n] - a2 * y[n];



Biquad IIR Low Pass Filter implementation

Example: Spresense_FrontEnd_Biquad_LPF.ino

Initialization of Biquad filter

```
arm_biquad_cascade_df2T_instance_f32 S;
const int numStages = 1;
float pCoeffs[5*numStages];
float pState[2*numStages];
…
void initializeBiquadLPF() {
  const float Q = 0.7;
  const uint32_t CUTTOFF_FREQ_HZ = 4000;
  float Wc = 2.*PI*CUTTOFF_FREQ_HZ/AS_SAMPLINGRATE_48000;

  float Alpha = arm_sin_f32(Wc)/(2.*Q);

  float numerator = 1.-arm_cos_f32(Wc);
  float b10 = numerator/2.;
  float b11 = numerator;
  float b12 = numerator/2.;
  float a10 = 1. + Alpha;
  float a11 = -2.*arm_cos_f32(Wc);
  float a12 = 1. - Alpha;

  pCoeffs[0] =  b10/a10;
  pCoeffs[1] =  b11/a10;
  pCoeffs[2] =  b12/a10;
  pCoeffs[3] = -a11/a10;
  pCoeffs[4] = -a12/a10;                                coefficient calculation

  arm_biquad_cascade_df2T_init_f32(&S, numStages, pCoeffs, pState);
}                                                       structure initialization
```

Implementation of the “signal_processing” function

```
void signal_process(int16_t* mono_input, int16_t* stereo_output, uint32_t sample_size) {

  static float pSrc[SAMPLE_SIZE];
  static float pDst[SAMPLE_SIZE];
  q15_t* q15_mono = (q15_t*)mono_input;
  arm_q15_to_float(&q15_mono[0], &pSrc[0], SAMPLE_SIZE);
  arm_biquad_cascade_df2T_f32(&S, &pSrc[0], &pDst[0], SAMPLE_SIZE);
  arm_float_to_q15(&pSrc[0], &q15_mono[0], SAMPLE_SIZE);
  mono_input = (int16_t*)q15_mono;                      Applying the IIR Low Pass Filter

  /* clean up the output buffer */
  memset(stereo_output, 0, sizeof(int16_t)*sample_size*2);

  /* copy the signal to output buffer */
  for (int n = SAMPLE_SIZE-1; n >= 0; --n)  {
   stereo_output[n*2] = stereo_output[n*2+1] = mono_input[n];
  }

  return;
}
```

Biquad IIR High Pass Filter implementation

Example: Spresense_FrontEnd_Biquad_HPF.ino

Initialization of Biquad filter

```
arm_biquad_cascade_df2T_instance_f32 S;
const int numStages = 1;
float pCoeffs[5*numStages];
float pState[2*numStages];
…
void initializeBiquadHPF() {
  const float Q = 0.7;
  const uint32_t CUTTOFF_FREQ_HZ = 8000;
  float Wc = 2.*PI*CUTTOFF_FREQ_HZ/AS_SAMPLINGRATE_48000;

  float Alpha = arm_sin_f32(Wc)/(2.*Q);

  float numerator = 1.+arm_cos_f32(Wc);
  float b10 = numerator/2.;
  float b11 = -numerator;
  float b12 = numerator/2.;
  float a10 = 1. + Alpha;
  float a11 = -2.*arm_cos_f32(Wc);
  float a12 = 1. - Alpha;

  pCoeffs[0] =  b10/a10;
  pCoeffs[1] =  b11/a10;
  pCoeffs[2] =  b12/a10;
  pCoeffs[3] = -a11/a10;
  pCoeffs[4] = -a12/a10;
```

coefficient calculation

```
  arm_biquad_cascade_df2T_init_f32(&S, numStages, pCoeffs, pState);
}
```

Implementation of the “signal_processing” function

```
void signal_process(int16_t* mono_input, int16_t* stereo_output, uint32_t sample_size) {

  static float pSrc[SAMPLE_SIZE];
  static float pDst[SAMPLE_SIZE];
  q15_t* q15_mono = (q15_t*)mono_input;
  arm_q15_to_float(&q15_mono[0], &pSrc[0], SAMPLE_SIZE);
  arm_biquad_cascade_df2T_f32(&S, &pSrc[0], &pDst[0], SAMPLE_SIZE);
  arm_float_to_q15(&pSrc[0], &q15_mono[0], SAMPLE_SIZE);
  mono_input = (int16_t*)q15_mono;


  /* clean up the output buffer */
  memset(stereo_output, 0, sizeof(int16_t)*sample_size*2);

  /* copy the signal to output buffer */
  for (int n = SAMPLE_SIZE-1; n >= 0; --n)  {
    stereo_output[n*2] = stereo_output[n*2+1] = mono_input[n];
  }

  return;
}
```

Same as LPF implementation

Biquad IIR Band Pass Filter implementation

Example: Spresense_FrontEnd_Biquad_HPF.ino

Initialization of Biquad filter

```
arm_biquad_cascade_df2T_instance_f32 S;
const int numStages = 1;
float pCoeffs[5*numStages];
float pState[2*numStages];
…
void initializeBiquadBPF() {
  const uint32_t CUTTOFF_FREQ_HZ = 8000;
  float Wc = 2.*PI*CUTTOFF_FREQ_HZ/AS_SAMPLINGRATE_48000;
  float Octave = 1./3.;
  float Bandwidth = (pow(2., Octave) - 1.)/pow(2., Octave/2);
  float Alpha = sin(Wc)*sinh(log(2.)/2.0*Bandwidth*Wc/sin(Wc));

  float numerator = 1.+arm_cos_f32(Wc);
  float b10 = Alpha.;
  float b11 = 0.;
  float b12 = -Alpha.;
  float a10 = 1. + Alpha;
  float a11 = -2.*arm_cos_f32(Wc);
  float a12 = 1. - Alpha;

  pCoeffs[0] =  b10/a10;
  pCoeffs[1] =  b11/a10;
  pCoeffs[2] =  b12/a10;
  pCoeffs[3] = -a11/a10;
  pCoeffs[4] = -a12/a10;
```

coefficient calculation

```
  arm_biquad_cascade_df2T_init_f32(&S, numStages, pCoeffs, pState);
}
```

Implementation of the "signal_processing" function

```
void signal_process(int16_t* mono_input, int16_t* stereo_output, uint32_t sample_size) {

  static float pSrc[SAMPLE_SIZE];
  static float pDst[SAMPLE_SIZE];
  q15_t* q15_mono = (q15_t*)mono_input;
  arm_q15_to_float(&q15_mono[0], &pSrc[0], SAMPLE_SIZE);
  arm_biquad_cascade_df2T_f32(&S, &pSrc[0], &pDst[0], SAMPLE_SIZE);
  arm_float_to_q15(&pSrc[0], &q15_mono[0], SAMPLE_SIZE);
  mono_input = (int16_t*)q15_mono;


  /* clean up the output buffer */
  memset(stereo_output, 0, sizeof(int16_t)*sample_size*2);

  /* copy the signal to output buffer */
  for (int n = SAMPLE_SIZE-1; n >= 0; --n)  {
    stereo_output[n*2] = stereo_output[n*2+1] = mono_input[n];
  }

  return;
}
```

Same as LPF implementation

Biquad IIR Notch Filter implementation

Example: Spresense_FrontEnd_Biquad_Notch.ino

Initialization of Biquad filter

```
arm_biquad_cascade_df2T_instance_f32 S;
const int numStages = 1;
float pCoeffs[5*numStages];
float pState[2*numStages];
…
void initializeBiquadBPF() {
  const uint32_t CUTTOFF_FREQ_HZ = 8000;
  float Wc = 2.*PI*CUTTOFF_FREQ_HZ/AS_SAMPLINGRATE_48000;
  float Octave = 1./10.;
  float Bandwidth = (pow(2., Octave) - 1.)/pow(2., Octave/2);
  float Alpha = sin(Wc)*sinh(log(2.)/2.0*Bandwidth*Wc/sin(Wc));

  float b10 =  1.;
  float b11 = -2.*arm_cos_f32(Wc).;
  float b12 =  1.;
  float a10 = 1. + Alpha;
  float a11 = -2.*arm_cos_f32(Wc);
  float a12 = 1. - Alpha;

  pCoeffs[0] =   b10/a10;
  pCoeffs[1] =   b11/a10;
  pCoeffs[2] =   b12/a10;
  pCoeffs[3] = -a11/a10;
  pCoeffs[4] = -a12/a10;                coefficient calculation

  arm_biquad_cascade_df2T_init_f32(&S, numStages, pCoeffs, pState);
}
```

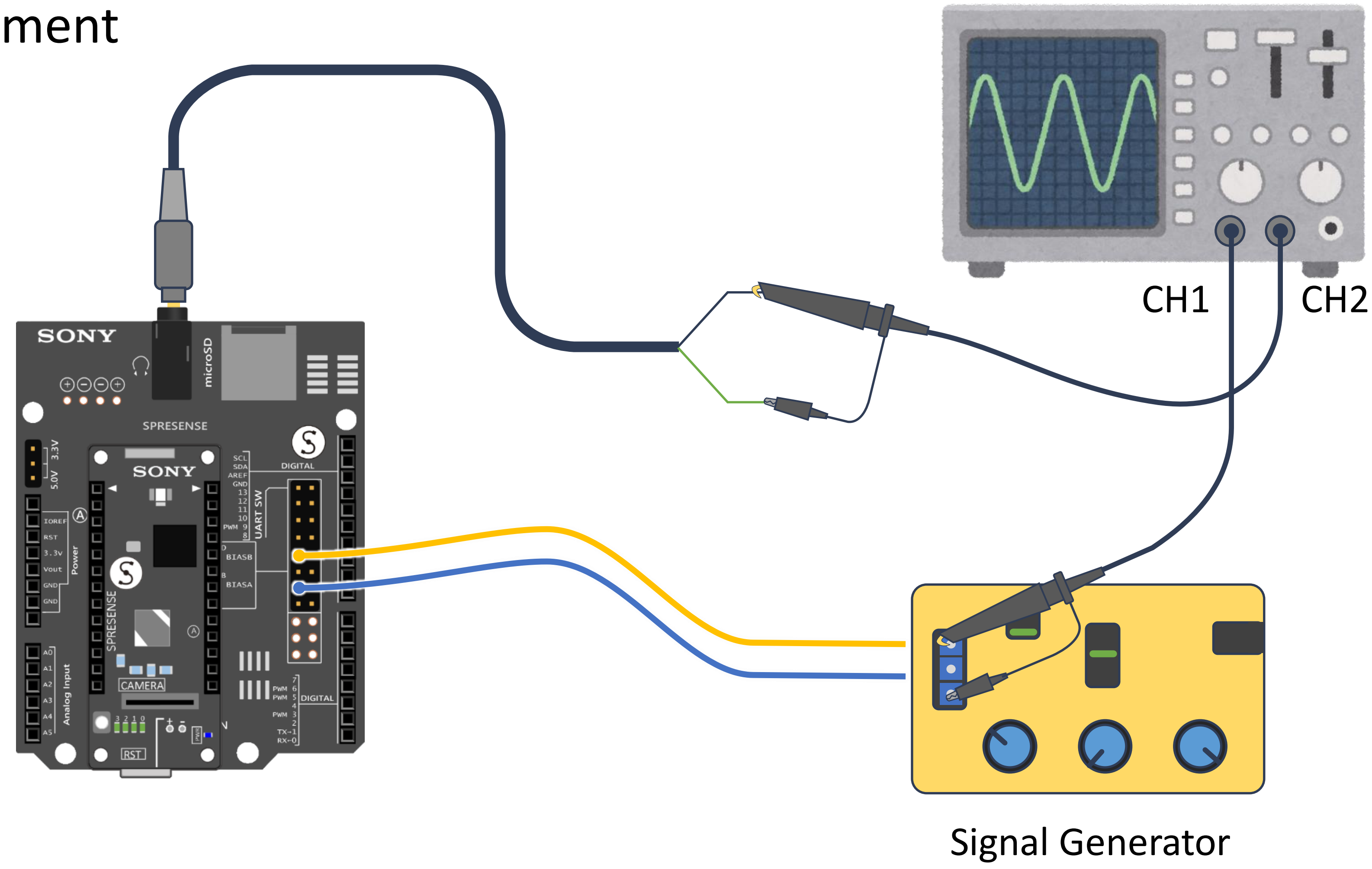
Implementation of the “signal_processing” function

```
void signal_process(int16_t* mono_input, int16_t* stereo_output, uint32_t sample_size) {

  static float pSrc[SAMPLE_SIZE];
  static float pDst[SAMPLE_SIZE];
  q15_t* q15_mono = (q15_t*)mono_input;
  arm_q15_to_float(&q15_mono[0], &pSrc[0], SAMPLE_SIZE);
  arm_biquad_cascade_df2T_f32(&S, &pSrc[0], &pDst[0], SAMPLE_SIZE);
  arm_float_to_q15(&pSrc[0], &q15_mono[0], SAMPLE_SIZE);
  mono_input = (int16_t*)q15_mono;


  /* clean up the output buffer */
  memset(stereo_output, 0, sizeof(int16_t)*sample_size*2);

  /* copy the signal to output buffer */
  for (int n = SAMPLE_SIZE-1; n >= 0; --n)  {
    stereo_output[n*2] = stereo_output[n*2+1] = mono_input[n];
  }

  return;
}
```

Same as LPF implementation

Operation Test for Biquad IIR Filter on Spresense

Test Environment



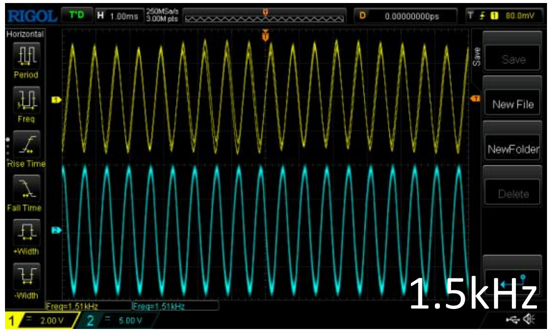
Operation Test for Biquad IIR Filter on Spresense

Biquad LPF Q:1.5

f_s: $48000Hz$ f_c: $1000Hz$

Biquad HPF Q:1.5

f_s: $48000Hz$ f_c: $1000Hz$



Biquad BPF $Bandwidth$: 1 $oct.$

f_s: $48000Hz$ f_c: $1500Hz$

Biquad Notch $Bandwidth$: 0.1 $oct.$

f_s: $48000Hz$ f_l: $1500Hz$

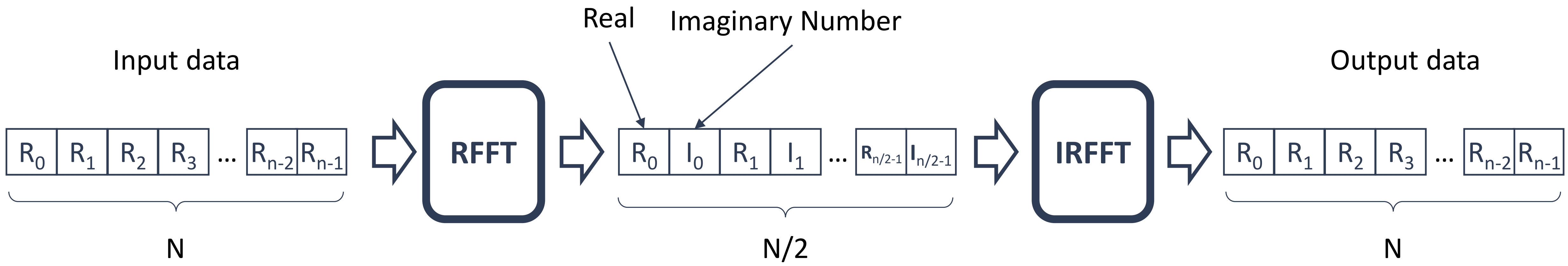
FFT implementation with ARM CMSIS

arm_status arm_rfft_fast_init_f32 (**const arm_cfft_instance_f32 * S,** **uint16_t fftLen** **)**	[in,out] S [in] fftLen	points to an arm_rfft_fast_instance_f32 structure length of the Real Sequence (number of samples)

void arm_rfft_fast_f32 (**arm_rfft_fast_instance_f32 * S,** **float32_t * p,** **float32_t * pOut,** **uint8_t ifftFlag** **)**	[in, out] S [in] p [out] pOut [in] ifftFlag	points to an arm_rfft_fast_instance_f32 structure points to input buffer (Source buffer is modified by this function.) points to output buffe value = 0: RFFT value = 1: RIFFT

FFT implementation with ARM CMSIS

Notes for using RFFT



After FFT conversion, real and imaginary data are combined, and the number of data is halved.

FFT (Fast Fourier Transform) implementation

Example: Spresense_FrontEnd_STFT.ino (Pass Through Implementation)

Initialization of Biquad filter

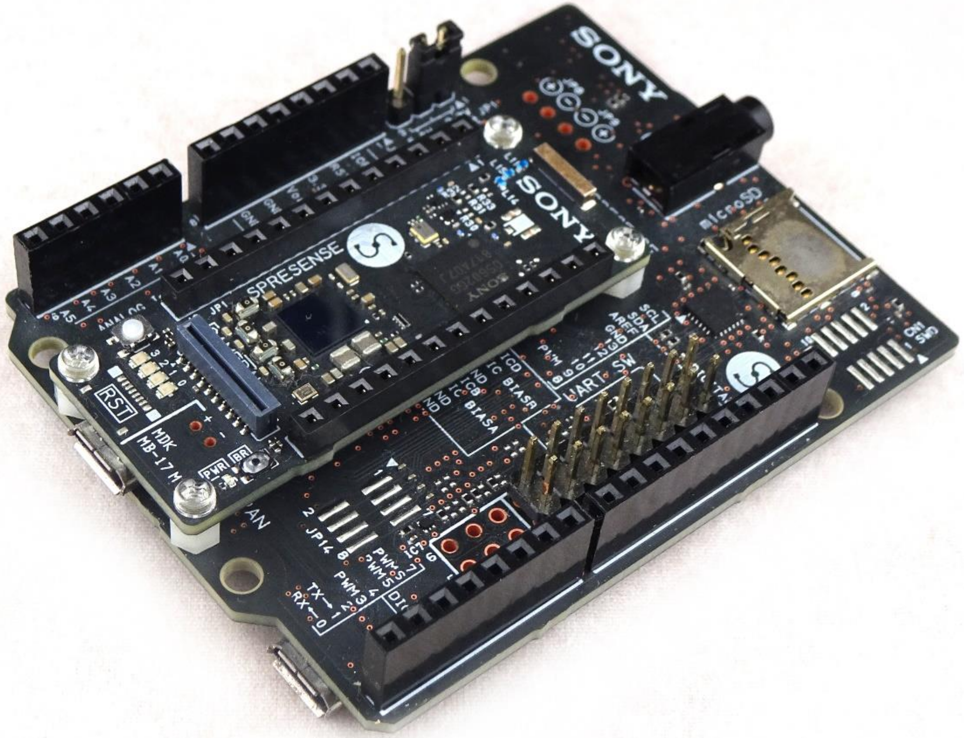
```
#define SAMPLE_SIZE 1024
arm_rfft_fast_instance_f32 S;
…
void setup() {
  …
  arm_rfft_fast_init_f32(&S, SAMPLE_SIZE);
  …
}
```

Initialize the structure

Implementation of the "signal_processing" function

```
void signal_process(int16_t* mono_input, int16_t* stereo_output, uint32_t sample_size) {

  static float pTmp[SAMPLE_SIZE];
  static float p1[SAMPLE_SIZE];

  q15_t* q15_mono = (q15_t*)mono_input;
  arm_q15_to_float(&q15_mono[0], &pTmp[0], SAMPLE_SIZE);
  arm_rfft_fast_f32(&S, &pTmp[0], &p1[0], 0);
  // TODO: Add some effects
  arm_rfft_fast_f32(&S, &p1[0], &pTmp[0], 1);
  arm_float_to_q15(&pTmp[0], &q15_mono[0], SAMPLE_SIZE);
  mono_input = (int16_t*)q15_mono;

  /* clean up the output buffer */
  memset(stereo_output, 0, sizeof(int16_t)*sample_size*2);
  /* copy the signal to output buffer */
  for (int n = SAMPLE_SIZE-1; n >= 0; --n)  {
    stereo_output[n*2] = stereo_output[n*2+1] = mono_input[n];
  }
  return;
}
```

FFT and iFFT implementation

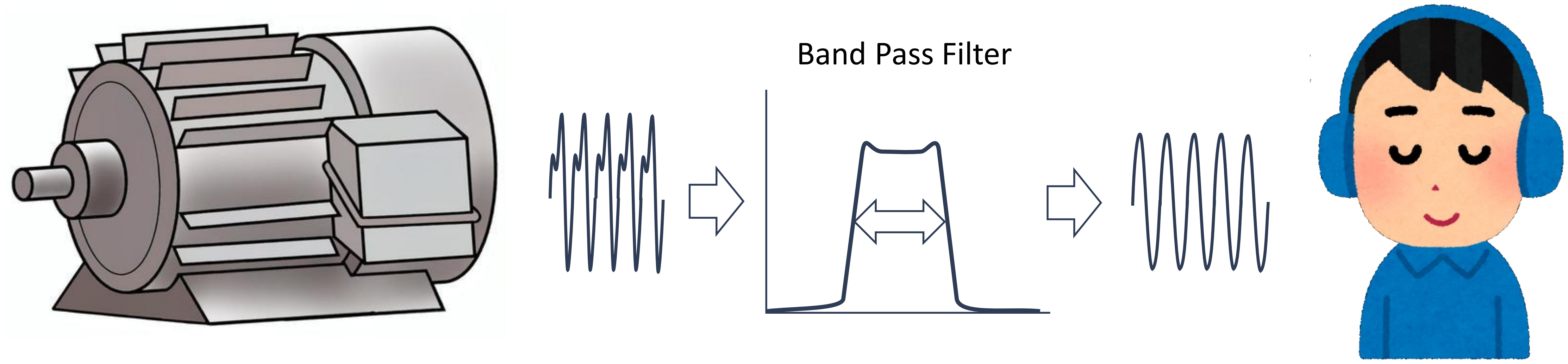


SPRESENSE

Real-Time Processing applications using Spresense

Specific frequency extraction by volumes

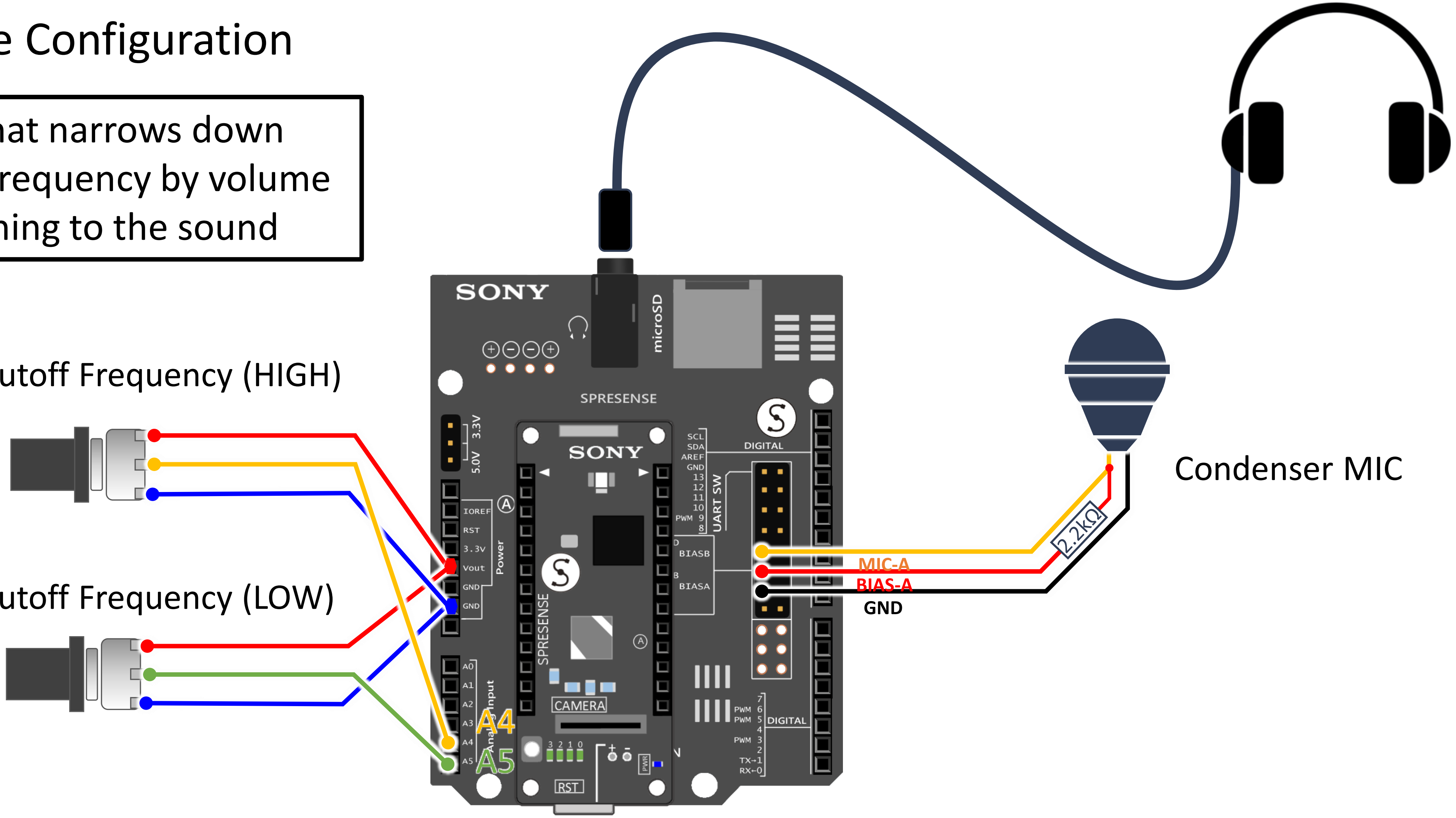
The bandwidth of the bandpass filter is changed in real time to identify the frequency band of abnormal sound.



Specific frequency extraction by volumes

Hardware Configuration

A device that narrows down the noise frequency by volume while listening to the sound



Specific frequency extraction by volumes

Note: Use A4 and A5 because A0-3 in Spresense are slow. If the processing still cannot be completed in time, consider moving the processing to a sub-core.

Example: Spresense_FrontEnd_FIR_BPF_VOL.ino

Setup coefficients of FIR Band Pass Filter

```
#define VOLUME_STEP (256)
...
void setupFirBPF(int high, int low) {
  static int high_ = 0;  static int low_  = VOLUME_STEP-1;

  if ((high_ == high) && (low_ == low)) return;
  high_ = high; low_ = low;

  const uint32_t freq_step = AS_SAMPLINGRATE_48000/2/VOLUME_STEP;
  uint32_t CUTTOFF_LOW_FREQ_HZ = low_*freq_step;
  uint32_t CUTTOFF_HIGH_FREQ_HZ = high_*freq_step;
  float Fl = (float)CUTTOFF_LOW_FREQ_HZ/AS_SAMPLINGRATE_48000;
  float Fh = (float)CUTTOFF_HIGH_FREQ_HZ/AS_SAMPLINGRATE_48000;
  const int H_TAPS = TAPS/2;
  int n = 0;
  for (int k = H_TAPS; k >= -H_TAPS; --k) {
   if (k == 0) pCoeffs[n] = 2.*(Fh - Fl);
   else         pCoeffs[n] = 2.*Fh*arm_sin_f32(2.*PI*Fh*k)/(2.*PI*Fh*k)
                            - 2.*Fl*arm_sin_f32(2.*PI*Fl*k)/(2.*PI*Fl*k);
   ++n;
  }

  for (int m = 0; m < TAPS; ++m)   pCoeffs[m] = (0.5 - 0.5*arm_cos_f32(2*PI*m/TAPS))*pCoeffs[m];

  arm_fir_init_f32(&S, TAPS, pCoeffs, pState, SAMPLE_SIZE);
}
```

BPF structure settings

Implementation of the “signal_processing” function

```
void signal_process(int16_t* mono_input, int16_t* stereo_output, uint32_t sample_size) {

  uint16_t V4 = analogRead(A4);
  uint16_t V5 = analogRead(A5);
  uint8_t v4 = map(V4, 0, 1023, 1, VOLUME_STEP-1);
  uint8_t v5 = map(V5, 0, 1023, 1, VOLUME_STEP-1);
```

Read Volumes value

```
  setupFirBPF(v4,v5);
```

Setup Bnad Pass Filter

```
  static float pSrc[SAMPLE_SIZE];
  static float pDst[SAMPLE_SIZE];
  q15_t* q15_mono = (q15_t*)mono_input;
  arm_q15_to_float(&q15_mono[0], &pSrc[0], SAMPLE_SIZE);
  arm_biquad_cascade_df2T_f32(&S, &pSrc[0], &pDst[0], SAMPLE_SIZE);
  arm_float_to_q15(&pSrc[0], &q15_mono[0], SAMPLE_SIZE);
  mono_input = (int16_t*)q15_mono;
```

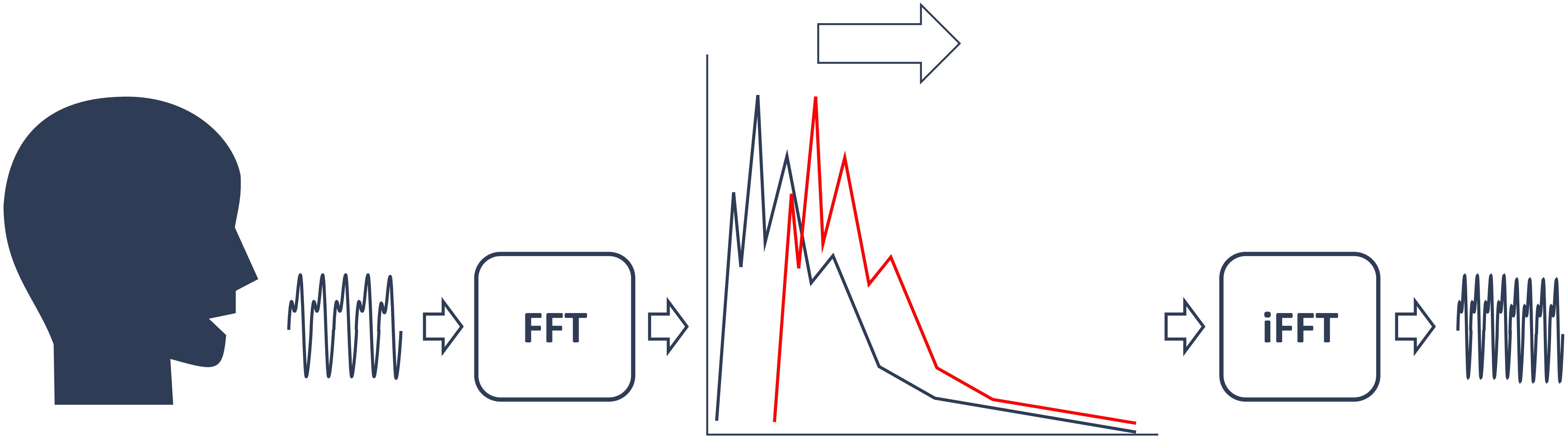
Applying FIR Band Pass Filter

```
  /* clean up the output buffer */
  memset(stereo_output, 0, sizeof(int16_t)*sample_size*2);

  /* copy the signal to output buffer */
  for (int n = SAMPLE_SIZE-1; n >= 0; --n)  {
   stereo_output[n*2] = stereo_output[n*2+1] = mono_input[n];
  }

  return;
}
```

Voice Changer Application

Basic Idea

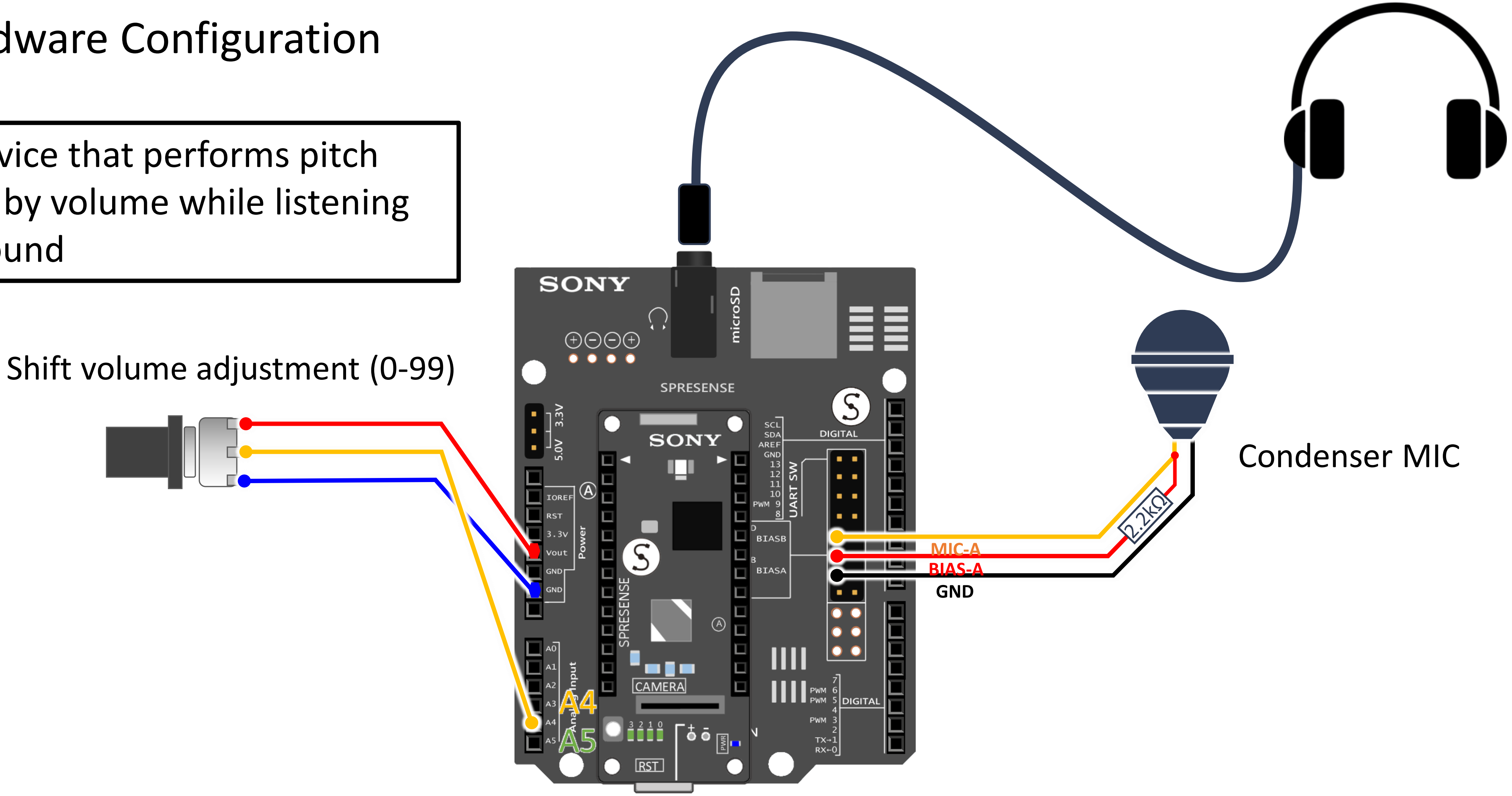
Changing the tone of voice
by shifting frequencies



Voice Changer Application

Hardware Configuration

A device that performs pitch shift by volume while listening to sound



Voice Changer Application

Example: Spresense_FrontEnd_STFT_voice_changer.ino

Implementation of the "signal_processing" function

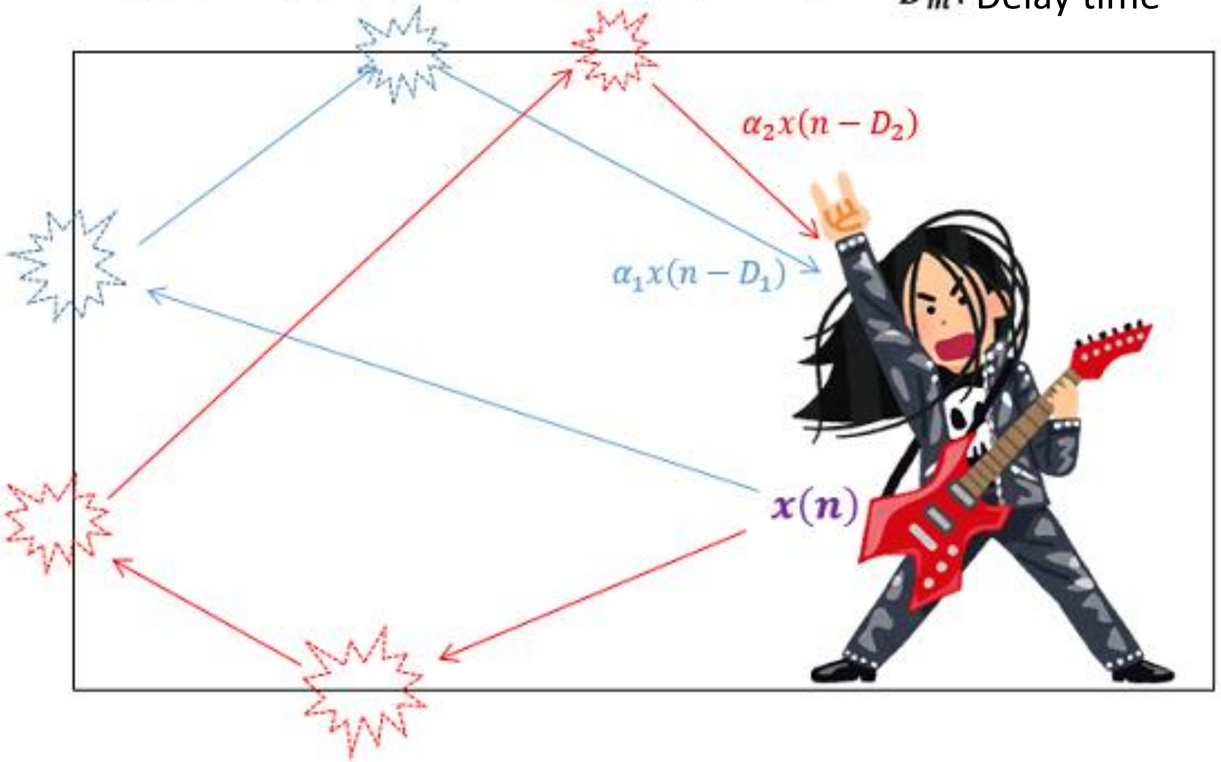
```
void signal_process(int16_t* mono_input, int16_t* stereo_output, uint32_t sample_size) {

  uint16_t V4 = analogRead(A4);
   int pitch_shift = map(V4, 0, 1023, 0, 99);
                                                          Volume Reading

  static float pTmp[SAMPLE_SIZE];
  static float p1[SAMPLE_SIZE];
  static float p2[SAMPLE_SIZE];

  q15_t* q15_mono = (q15_t*)mono_input;
  arm_q15_to_float(&q15_mono[0], &pTmp[0], SAMPLE_SIZE);
  arm_rfft_fast_f32(&S, &pTmp[0], &p1[0], 0);
  memcpy(&p2[pitch_shift*2], &p1[0], (SAMPLE_SIZE-pitch_shift)*2*sizeof(float));
  arm_cfft_f32(&S, &p2[0], &pTmp[0] 1);
  arm_float_to_q15(&pTmp[0], &q15_mono[0], SAMPLE_SIZE);
  mono_input = (int16_t*)q15_mono;        Shift spectrum according to volume value

  /* clean up the output buffer */
  memset(stereo_output, 0, sizeof(int16_t)*sample_size*2);
  /* copy the signal to output buffer */
  for (int n = SAMPLE_SIZE-1; n >= 0; --n)  {
    stereo_output[n*2] = stereo_output[n*2+1] = mono_input[n];
  }
  return;
}
```

Digital Effector (Echo)

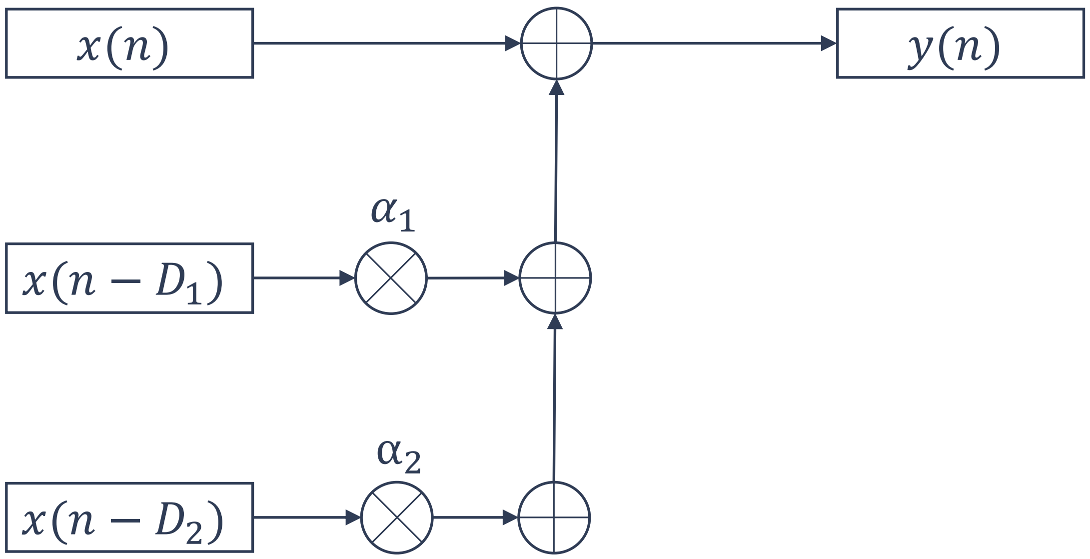
FIR Type of Effector

$$y(n) = x(n) + \alpha_1 x(n - D_1) + \alpha_2 x(n - D_2)$$

α_m: Attenuation
D_m: Delay time



Different from other implementations in that the delay times D1 and D2 are very large, so a large amount of buffers are required



Digital Effector (Echo)

Example: Spresense_FrontEnd_EchoEffect.ino

Implementation of the “signal_processing” function

```
#define SAMPLE_SIZE (720)

void signal_process(int16_t* mono_input, int16_t* stereo_output, uint32_t sample_size) {

  /* memory pool for 1.5sec (= 720*100*(1/48000)) */
  static const int lines = 100;
  static int16_t src_buf[SAMPLE_SIZE*lines];  /* 2*720*100=144kBytes */

  /* shift the buffer data in src_buf and add the latest data to top of the bufer */
  memcpy(&src_buf[0], &src_buf[SAMPLE_SIZE], SAMPLE_SIZE*sizeof(int16_t)*(lines-1));
  memcpy(&src_buf[(lines-1)*SAMPLE_SIZE], &mono_input[0], SAMPLE_SIZE*sizeof(int16_t));
```

Buffer for storing samples for 1.5 seconds

```
  /* set constatns for echo effect */
  static const uint32_t D1_in_ms = 300; /* milli sec */
  static const uint32_t D2_in_ms = 600; /* milli sec */
  static const uint32_t offset1 = D1_in_ms * 48000 / 1000;
  static const uint32_t offset2 = D2_in_ms * 48000 / 1000;
```

Delay value setting

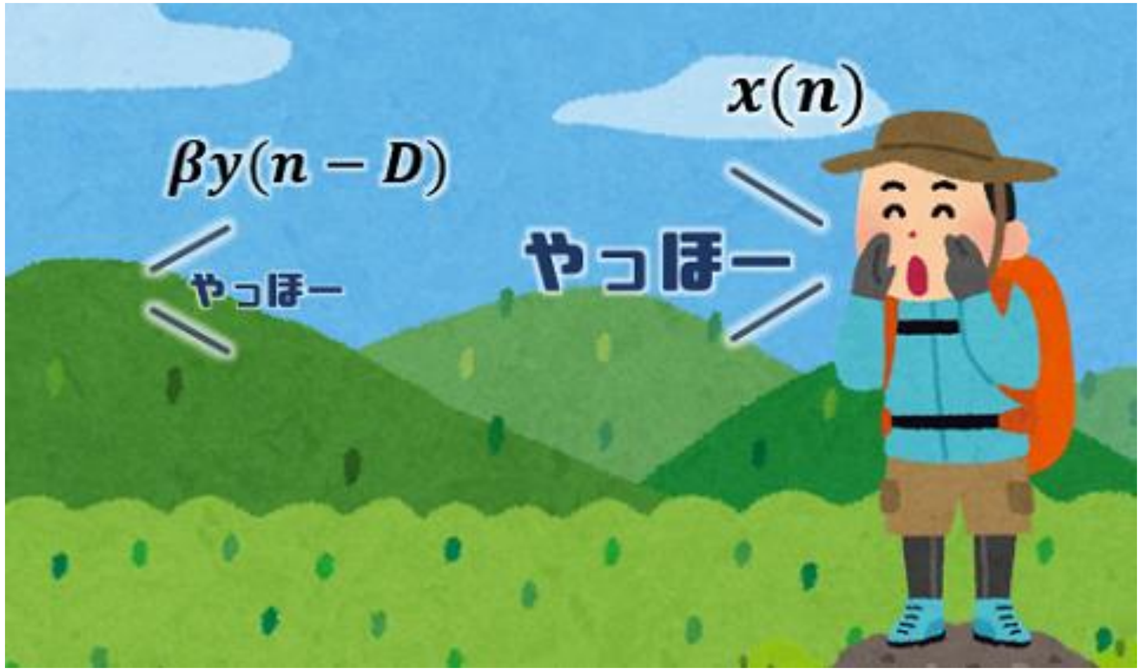
```
  const int src_buf_end_point = lines*SAMPLE_SIZE-1;
  for (int n = SAMPLE_SIZE-1; n >= 0; --n) {
    /* set h1 = 1/2, h2 = 1/4 to reduce calculation costs */
    mono_input[(SAMPLE_SIZE-1)-n] = src_buf[src_buf_end_point-n]
                                  + src_buf[src_buf_end_point-n-offset1]/2
                                  + src_buf[src_buf_end_point-n-offset2]/4;
  }
```

Applying Echo Effect

```
  /* clean up the output buffer */
  memset(stereo_output, 0, sizeof(int16_t)*sample_size*2);
  /* copy the signal to output buffer */
  for (int n = SAMPLE_SIZE-1; n >= 0; --n)  {
    stereo_output[n*2] = stereo_output[n*2+1] = mono_input[n];
  }
  return;
}
```

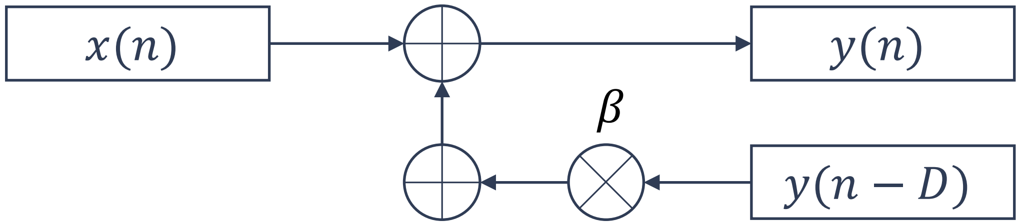
Digital Effector (Reverb)

IIR Type of Effector



As with echo, the delay time D is very large, so a large amount of buffer is required

$$y(n) = x(n) + \beta y(n - D)$$



Digital Effector (Reverb)

Example: Spresense_FrontEnd_ReverbEffect.ino

Implementation of the “signal_processing” function

```
#define SAMPLE_SIZE (720)

void signal_process(int16_t* mono_input, int16_t* stereo_output, uint32_t sample_size) {

/* memory pool for 1.5sec (= 720*100*(1/48000))  */
 static const int lines = 100;
 static int16_t out_buf[SAMPLE_SIZE*lines];  /* 2*720*100=144kBytes */
```

Buffer for storing output for 1.5 seconds

```
 /* set constatns for echo effect */
 static const uint32_t D_in_ms = 600; /* milli sec */
 static const uint32_t offset = D_in_ms * 48000 / 1000;
```

Delay value setting

```
 const int src_buf_end_point = lines*SAMPLE_SIZE-1;
 for (int n = SAMPLE_SIZE-1; n >= 0; --n) {
  /* set alpha = 1/2 to reduce calculation costs */
  mono_input[(SAMPLE_SIZE-1)-n] =  mono_input[(SAMPLE_SIZE-1)-n]
                                + out_buf[src_buf_end_point-n-offset]/2;
 }
```

Applying Reverb Effect

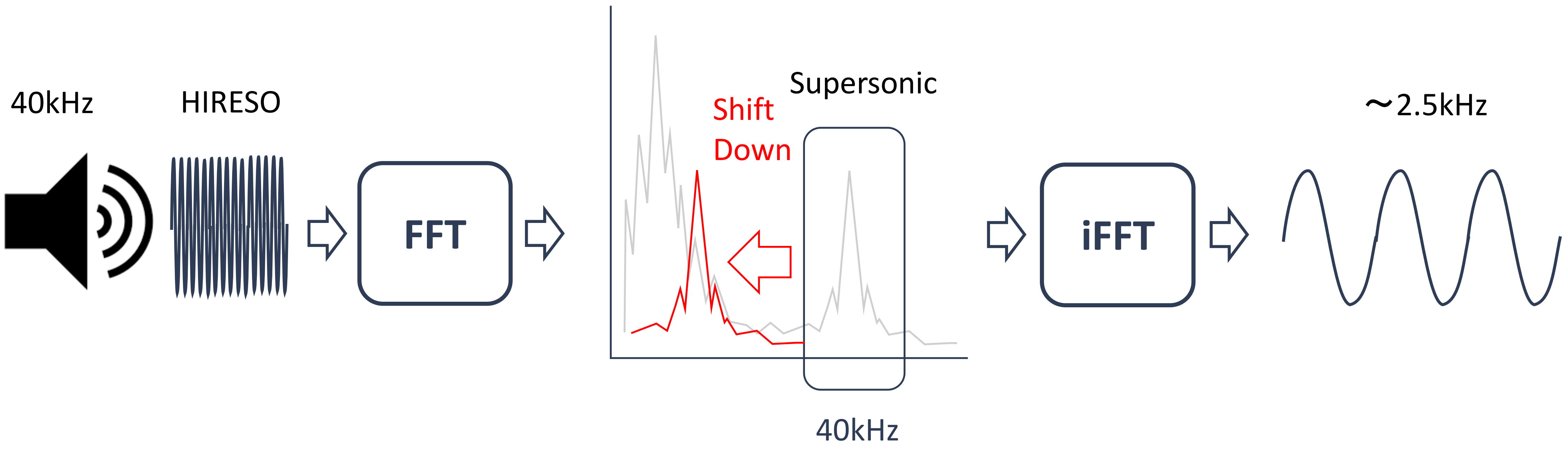
```
 /* shift the buffer data in src_buf and add the latest data to top of the bufer */
 memcpy(&out_buf[0], &out_buf[SAMPLE_SIZE], SAMPLE_SIZE*sizeof(int16_t)*(lines-1));
 memcpy(&out_buf[(lines-1)*SAMPLE_SIZE], &mono_input[0], SAMPLE_SIZE*sizeof(int16_t));
```

Add output data to storing buffer

```
  /* clean up the output buffer */
  memset(stereo_output, 0, sizeof(int16_t)*sample_size*2);
  /* copy the signal to output buffer */
  for (int n = SAMPLE_SIZE-1; n >= 0; --n)  {
   stereo_output[n*2] = stereo_output[n*2+1] = mono_input[n];
  }
  return;
}
```

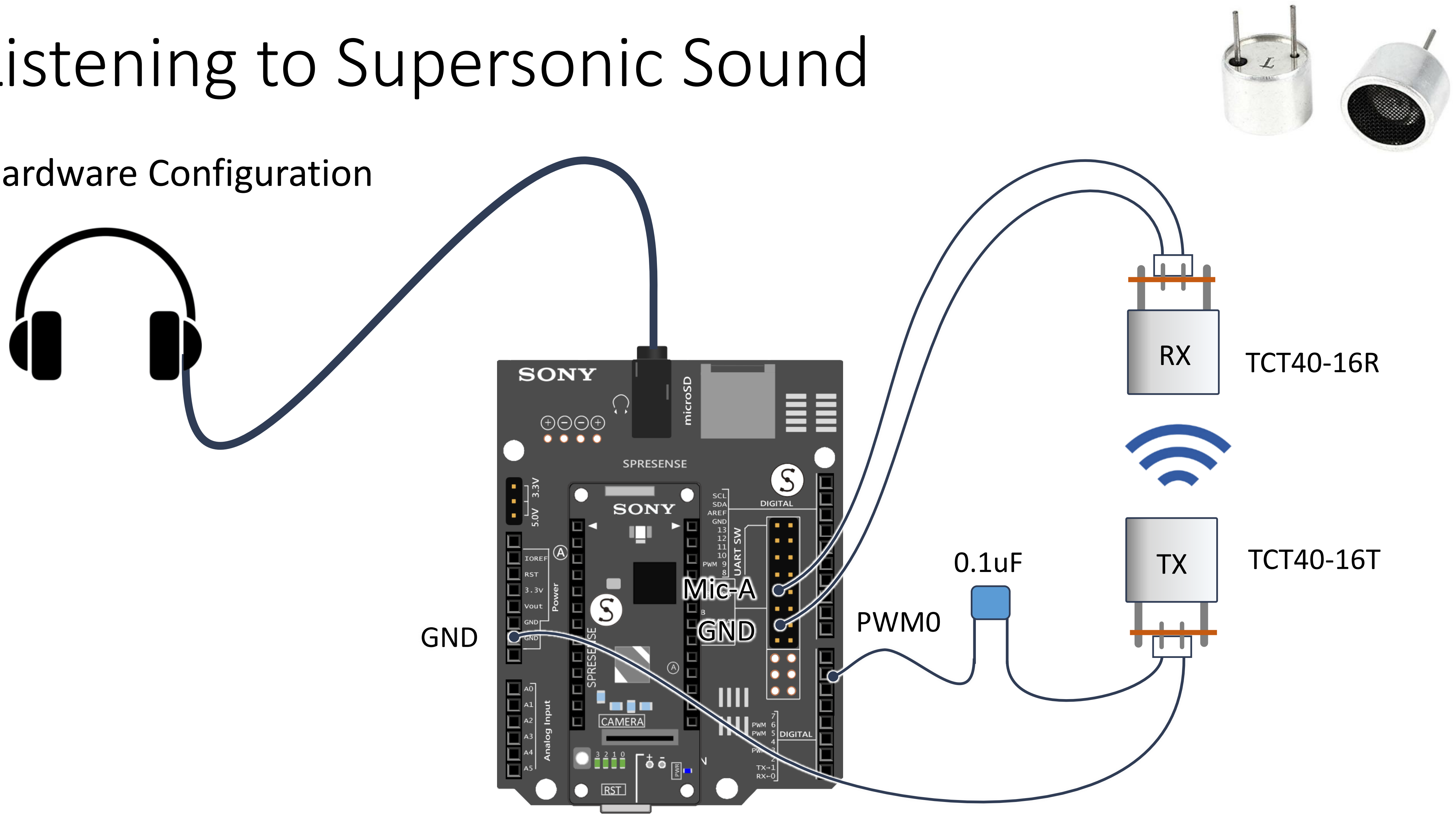
Listening to Supersonic Sound

Shift down supersonic spectrum to the audible range



Listening to Supersonic Sound

Hardware Configuration



Listening to Supersonic Sound

Note that the audio system must be set to high-resolution to capture supersonic waves

Example: Spresense_FrontEnd_ReverbEffect.ino

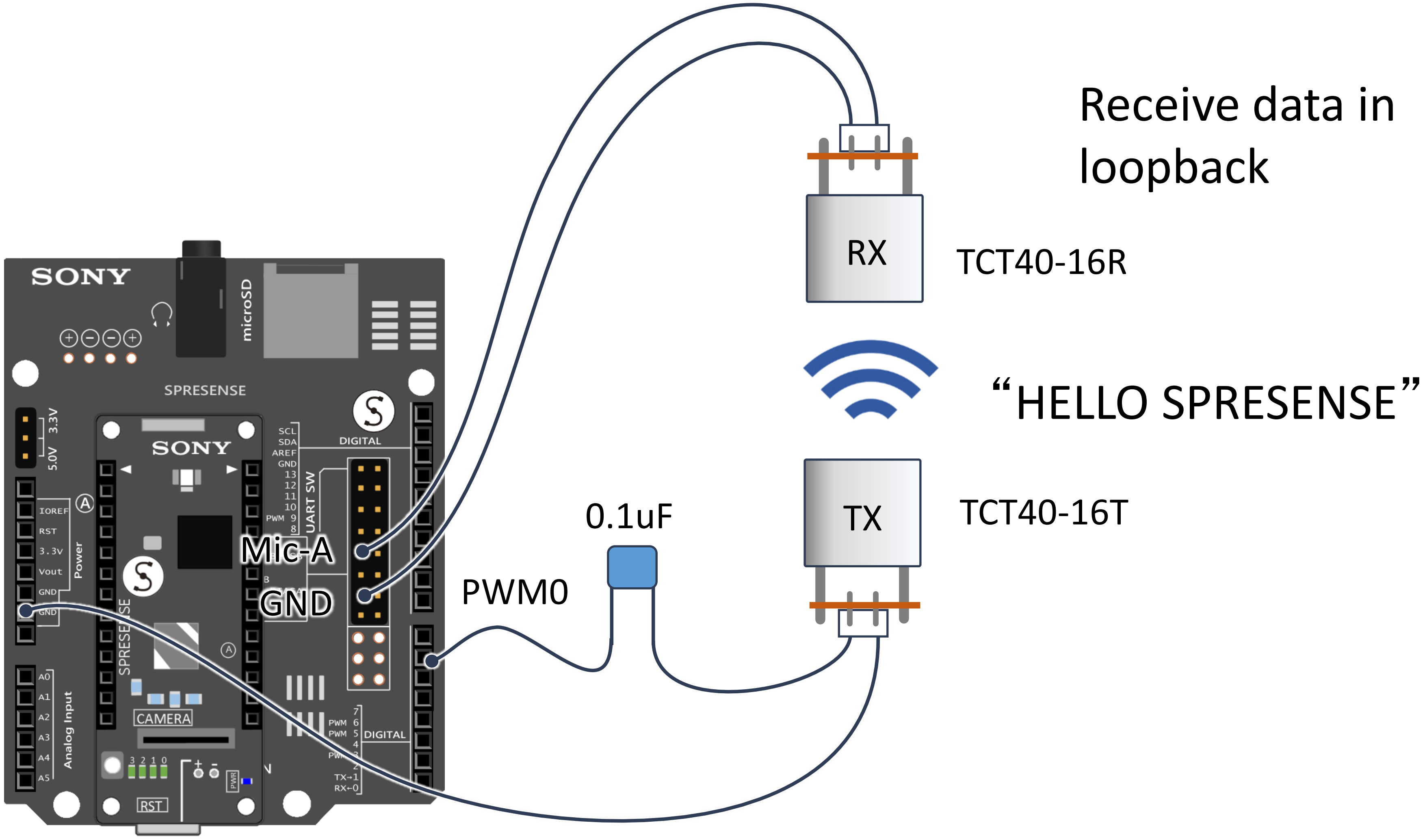
Setting PWM and High Resolution Audio (192kHz)

```
#include <sys/ioctl.h>
#include <stdio.h>
#include <fcntl.h>
#include <nuttx/timers/pwm.h>
int fd;
struct pwm_info_s info;

arm_rfft_fast_instance_f32 S;

void setup() {
  …
  arm_rfft_fast_init_f32(&S,  SAMPLE_SIZE);

  // PWM 40kHz                                    Setting for PWM
  fd = open("/dev/pwm0", O_RDONLY);
  info.frequency = 40000; // 40kHz
  info.duty      = 0x7fff; // duty 50:50
  ioctl(fd, PWMIOC_SETCHARACTERISTICS, (unsigned long)((uintptr_t)&info));
  ioctl(fd, PWMIOC_START, 0);
  …
  /* set clock mode */                            High Resolution Setting
  theFrontEnd->setCapturingClkMode(FRONTEND_CAPCLK_HIRESO);
  theMixer->setRenderingClkMode(OUTPUTMIXER_RNDCLK_HIRESO);
  …
}
```

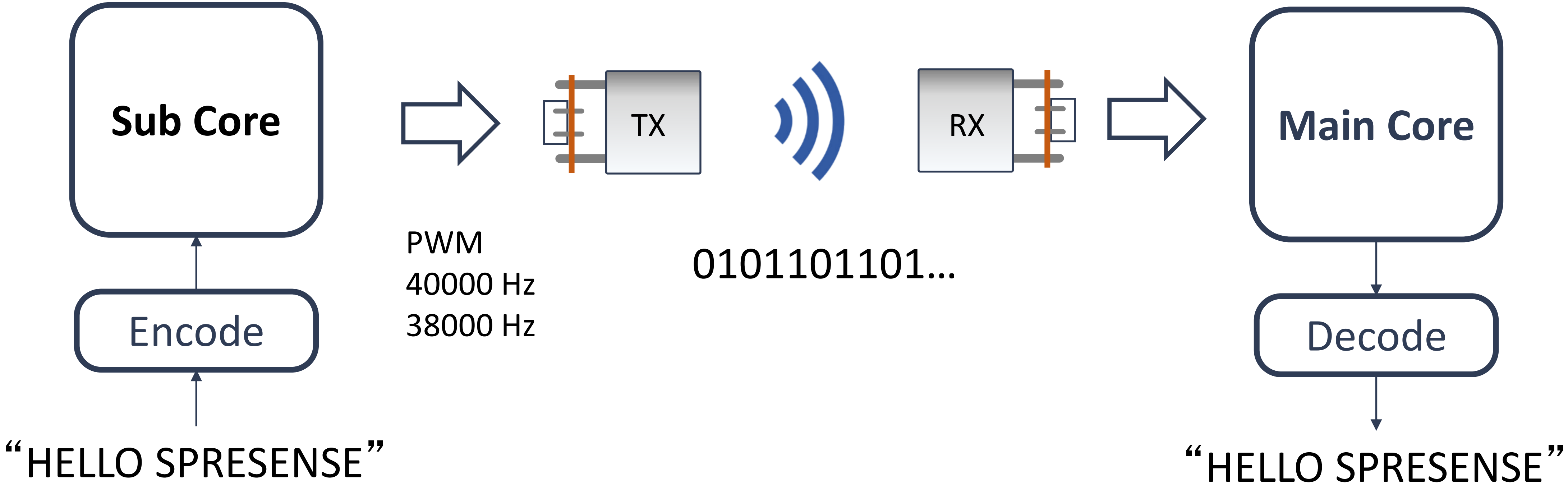
Implementation of the “signal_processing” function

```
void signal_process(int16_t* mono_input, int16_t* stereo_output, uint32_t sample_size) {
  uint32_t start_time = micros();
  static float pTmp[SAMPLE_SIZE];
  static float p1[SAMPLE_SIZE];
  static float p2[SAMPLE_SIZE];

  q15_t* q15_mono = (q15_t*)mono_input;                Shift Spectrum  38.5kHz down
  arm_q15_to_float(&q15_mono[0], &pTmp[0], SAMPLE_SIZE);
  arm_rfft_fast_f32(&S,  &pTmp[0], &p1[0], 0);
  int shift = 200;                          /*19200/1024*200 = 38500Hz shift */
  memcpy(&p2[0], &p1[shift*2], (SAMPLE_SIZE/2-shift)*sizeof(float));    /* low pitch */
  arm_rfft_fast_f32(&S, &p2[0], &pTmp[0], 1);
  arm_float_to_q15(&pTmp[0], &q15_mono[0], SAMPLE_SIZE);
  mono_input = (int16_t*)q15_mono;

  /* clean up the output buffer */
  memset(stereo_output, 0, sizeof(int16_t)*sample_size*2);
  /* copy the signal to output buffer */
  for (int n = SAMPLE_SIZE-1; n >= 0; --n)  {
   stereo_output[n*2] = stereo_output[n*2+1] = mono_input[n];
  }
  uint32_t duration = micros() - start_time;
  Serial.println("process time = " + String(duration));
  return;
}
```

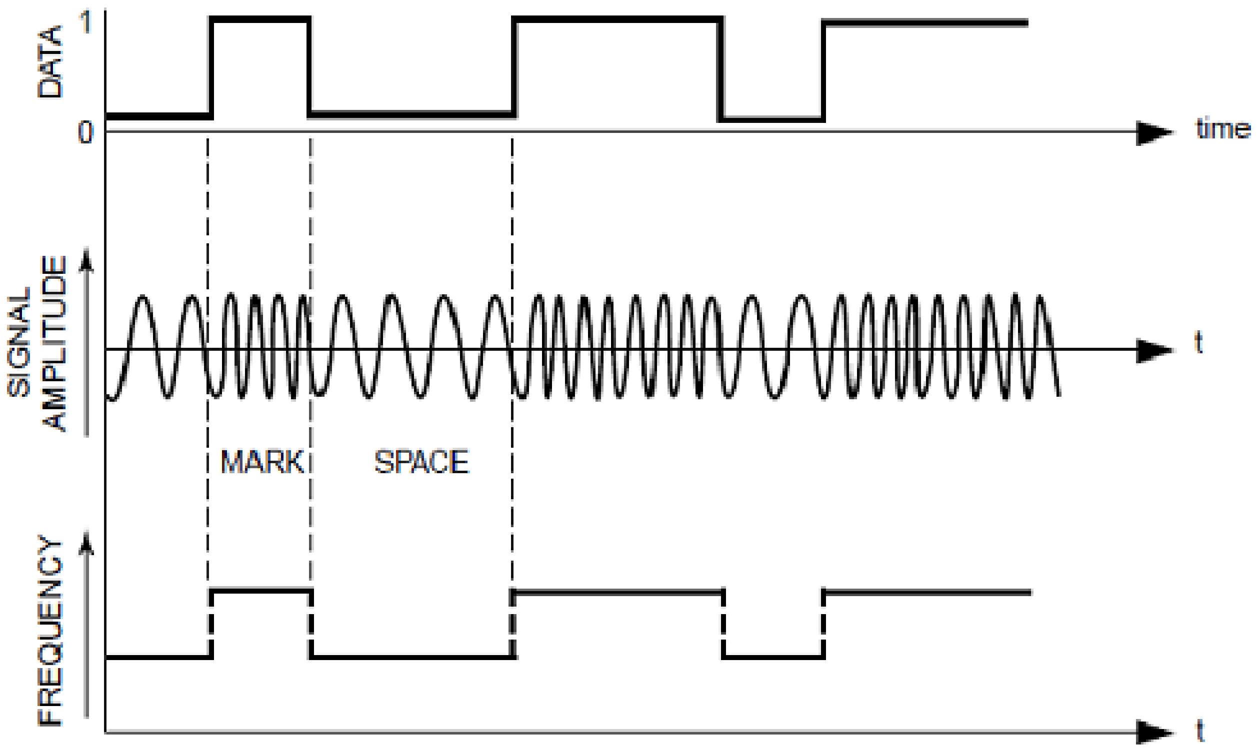
Super Sonic Communication



Super Sonic Communication



Super Sonic Communication

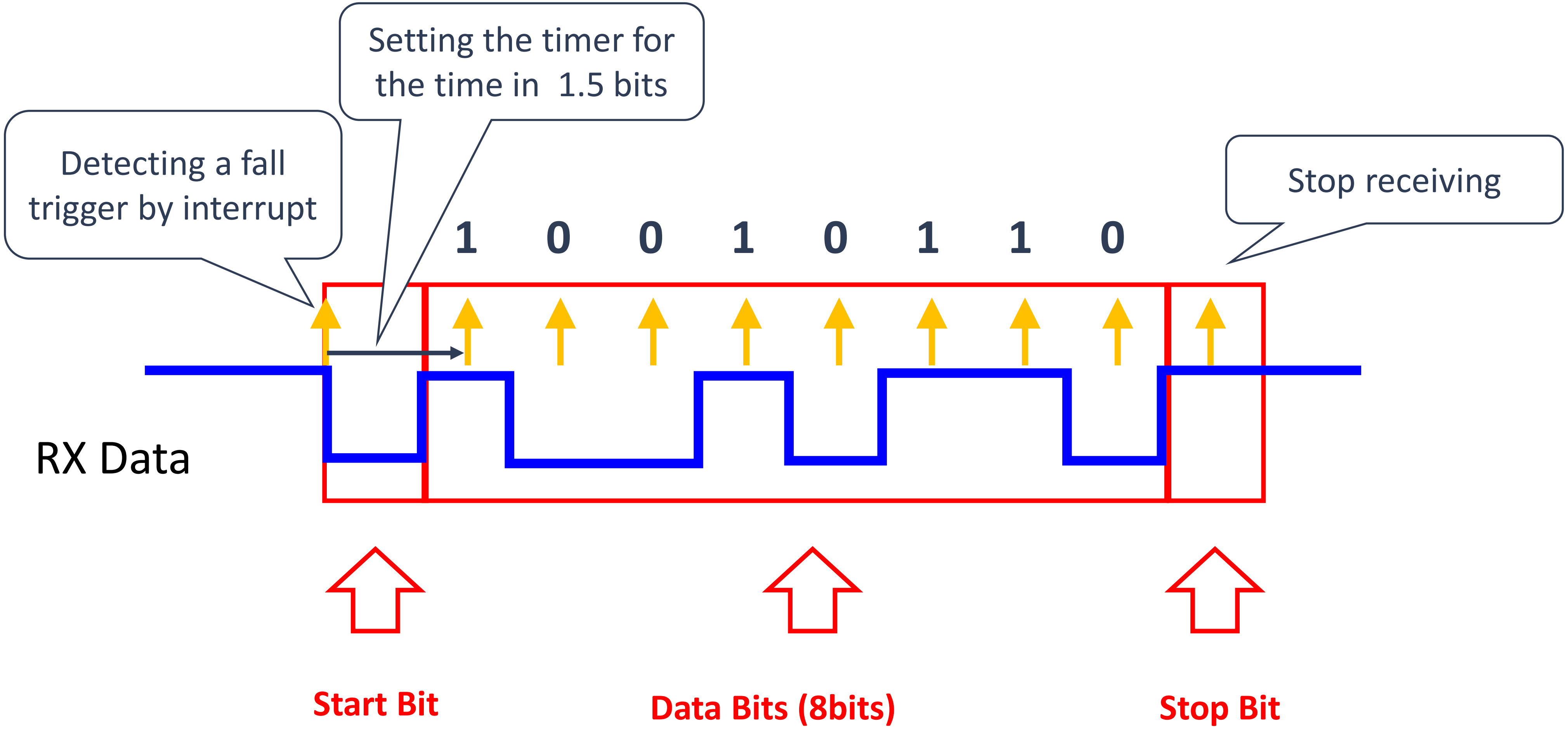


FSK (Frequency modulation method)
is used as the communication method.

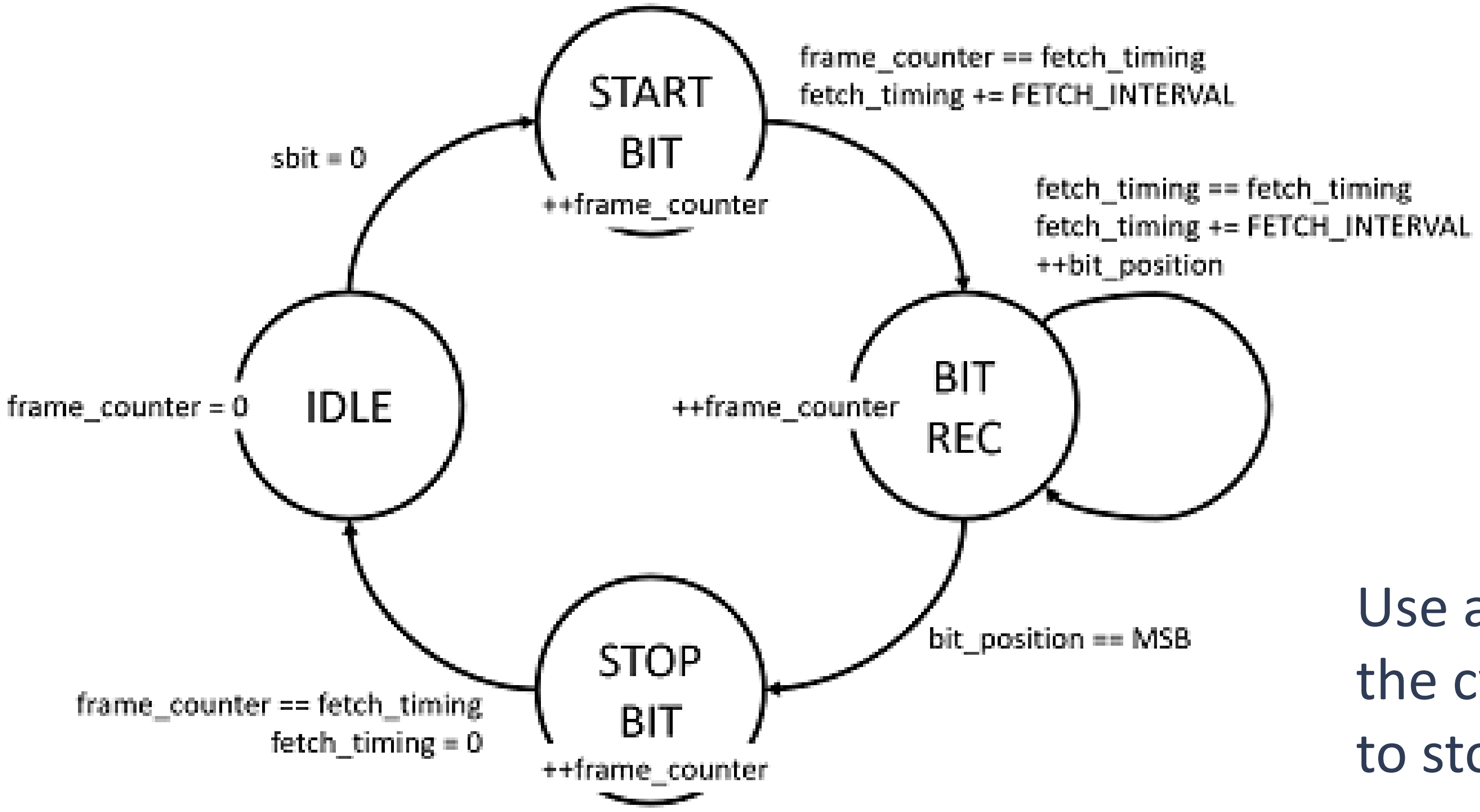
Set MARK/SPACE to 40 kHz and 38 kHz.

MARK [1] : 40000Hz
SPACE [0] : 38000Hz

Super Sonic Communication



Super Sonic Communication



Use a state machine to manage the cycle from start bit detection to stop bit.

Super Sonic Communication

Example: Spresense_supersonic_communicator/SubTX/SubTX.ino

Transmission sub-core implementation

```
#include <MP.h>
#include <sys/ioctl.h>
#include <stdio.h>
#include <fcntl.h>
#include <nuttx/timers/pwm.h>
int fd;
struct pwm_info_s info;

const int SPACE = 38000;
const int MARK  = 40000;

const int SAMPLE_SIZE = 1024;
const int SAMPLING_RATE = 192000;
const int DelayMicros = SAMPLE_SIZE*4*1000000/SAMPLING_RATE;

void send_signal(uint16_t output_hz) {
  info.frequency = output_hz;
  info.duty      = 0x7fff;  // duty 50:50
  ioctl(fd, PWMIOC_SETCHARACTERISTICS, (unsigned long)((uintptr_t)&info));
  ioctl(fd, PWMIOC_START, 0);
  delayMicroseconds(DelayMicros);
}
```

modulation processing

```
void setup() {
  MP.begin();
  fd = open("/dev/pwm0", O_RDONLY);
  info.frequency = MARK;
  info.duty      = 0x7fff;
  ioctl(fd, PWMIOC_SETCHARACTERISTICS, (unsigned long)((uintptr_t)&info));
  ioctl(fd, PWMIOC_START, 0);
}
```

PWM Settings

```
void encode(uint8_t c) {
  send_signal(SPACE);  // send start bit
  for (int n = 0; n < 8; ++n, c = c >> 1) { /* LSB first */
    if (c & 0x01) send_signal(MARK);    /* mark (1) */
    else          send_signal(SPACE);   /* space (0) */
  }
  send_signal(MARK);  // send stop bit
}
```

Decompose character into bits

```
void loop() {
  const char* const str = "HELLO SPRESENSE¥n";
  int n = strlen(str);
  for (int i = 0; i < n; ++i) {  encode(str[i]);  }
  delay(100);
}
```

Sends "HELLO SPRESENSE" every 100 milliseconds

Super Sonic Communication

Example: Spresense_supersonic_communicator/MainRX/MainRX.ino

Receiving main core implementation (1)

```
#include <FrontEnd.h>
#include <MemoryUtil.h>
#include <arch/board/board.h>
#include <MP.h>

#define SAMPLE_SIZE (1024)
FrontEnd *theFrontEnd;

const int32_t channel_num = AS_CHANNEL_MONO;
const int32_t bit_length  = AS_BITLENGTH_16;
const int32_t sample_size = SAMPLE_SIZE;
const int32_t frame_size  = sample_size * (bit_length / 8) * channel_num;
bool isErr = false;


#define ARM_MATH_CM4
#define __FPU_PRESENT 1U
#include <arm_math.h>
arm_rfft_fast_instance_f32 S;
```

MACROs for State Management

```
#define IDLE_STATE     (0)
#define STARTBIT_STATE (1)
#define BITREC_STATE   (2)
#define STOPBIT_STATE  (3)
#define FETCH_INTERVAL (4)
#define MSBBIT_INDEX   (7)
```

```
const int SPACE = 38000;
const int MARK  = 40000;
```

Variables for State Management

```
static uint8_t frame_cnt = 0;
static uint8_t fetch_timing = 1;
static uint8_t bpos = 0;
static uint8_t cur_state = IDLE_STATE;
static char    output = 0;
```

Function for IDLE state processing

```
void idle_phase(uint8_t sbit) {
  if (sbit == 0)  cur_state = STARTBIT_STATE;
  frame_cnt = 0;  fetch_timing = 1;
  output = 0;
  return;
}
```

Function for STARTBIT state processing

```
void startbit_phase(uint8_t sbit) {
  ++frame_cnt;
  if (frame_cnt != fetch_timing) return;
  cur_state = BITREC_STATE;
  fetch_timing += FETCH_INTERVAL;
  return;
}
```

Function for BITREC state processing

```
void bitrec_phase(uint8_t sbit) {
if (++frame_cnt != fetch_timing) return;
  output = output | (sbit << bpos);
  fetch_timing += FETCH_INTERVAL;
  if (++bpos > MSBBIT_INDEX)   cur_state = STOPBIT_STATE;
  return;
}
```

Super Sonic Communication

Example: Spresense_supersonic_communicator/MainRX/MainRX.ino

Receiving main core implementation (1)

Functions for STOPBIT state processing

```
bool stopbit_phase(uint8_t sbit) {
  if (++frame_cnt != fetch_timing) return;
  Serial.write(output);  // interim implementation
  frame_cnt = 0; bpos = 0;
  cur_state = IDLE_STATE;
  return;
}
....

static void frontend_pcm_cb(AsPcmDataParam pcm) {
  static uint8_t mono_input[frame_size];
  static const bool time_measurement = false;

  frontend_signal_input(pcm, mono_input, frame_size);
  signal_process((int16_t*)mono_input, (int16_t*)stereo_output, sample_size);
  return;
}

void frontend_signal_input(AsPcmDataParam pcm, uint8_t* input, uint32_t frame_size) {
  /* clean up the input buffer */
  memset(input, 0, frame_size);

  /* copy the signal to signal_input buffer */
  if (pcm.size != 0)   memcpy(input, pcm.mh.getPa(), pcm.size);
}
```

Peak detection by FFT · **MARK/SPACE judgment** · **State Machine**

```
void signal_process(int16_t* mono_input, int16_t* stereo_output, uint32_t sample_size) {
  uint32_t start_time = micros();
  static float pSrc[SAMPLE_SIZE];
  static float pDst[SAMPLE_SIZE];
  static float tmpBuf[SAMPLE_SIZE];
  float maxValue;
  uint32_t index;
  const float df = AS_SAMPLINGRATE_192000/SAMPLE_SIZE;

  arm_q15_to_float(&mono_input[0], &pSrc[0], SAMPLE_SIZE);
  arm_rfft_fast_f32(&S, &pSrc[0], &tmpBuf[0], 0);
  arm_cmplx_mag_f32(&tmpBuf[0], &pDst[0], SAMPLE_SIZE / 2);
  arm_max_f32(&pDst[0], SAMPLE_SIZE/2, &maxValue, &index);

  float peakFs = (float)index*df;
  const int fc = ((SPACE + MARK)/2) / df; // 39kHz

  uint8_t sbit;
  if (index < fc) sbit = 0;
  else if (index > fc) sbit = 1;
  switch(cur_state) {
   case IDLE_STATE:   idle_phase(sbit); break;
   case STARTBIT_STATE:  startbit_phase(sbit); break;
   case BITREC_STATE: bitrec_phase(sbit); break;
   case STOPBIT_STATE:   stopbit_phase(sbit); break;
  }

  return;
}
```

Super Sonic Communication

Example: Spresense_supersonic_communicator/MainRX/MainRX.ino

Receiving main core implementation (2)

```
void setup() {
  const int subcore = 1;
  Serial.begin(115200);
  MP.begin(subcore);
  arm_rfft_fast_init_f32(&S, SAMPLE_SIZE);

  initMemoryPools();
  createStaticPools(MEM_LAYOUT_RECORDINGPLAYER);

  theFrontEnd = FrontEnd::getInstance();

  theFrontEnd->setCapturingClkMode(FRONTEND_CAPCLK_HIRESO);

  theFrontEnd->begin(frontend_attention_cb);
  theFrontEnd->setMicGain(0);
  theFrontEnd->activate(frontend_done_cb);
  delay(100); /* waiting for Mic startup */
  AsDataDest dst;
  dst.cb = frontend_pcm_cb;
  theFrontEnd->init(channel_num, bit_length, sample_size, AsDataPathCallback, dst);
  Serial.println("Setup: FrontEnd initialized");

  theFrontEnd->start();

}
```

High Resolution Setting

No MIXER setting as there is no output

```
void loop() {
  if (isErr == true) {
    board_external_amp_mute_control(true);
    theFrontEnd->stop();
    theFrontEnd->deactivate();
    theFrontEnd->end();
    Serial.println("Capturing Process Terminated");
    while(1) {};
  }
}
```

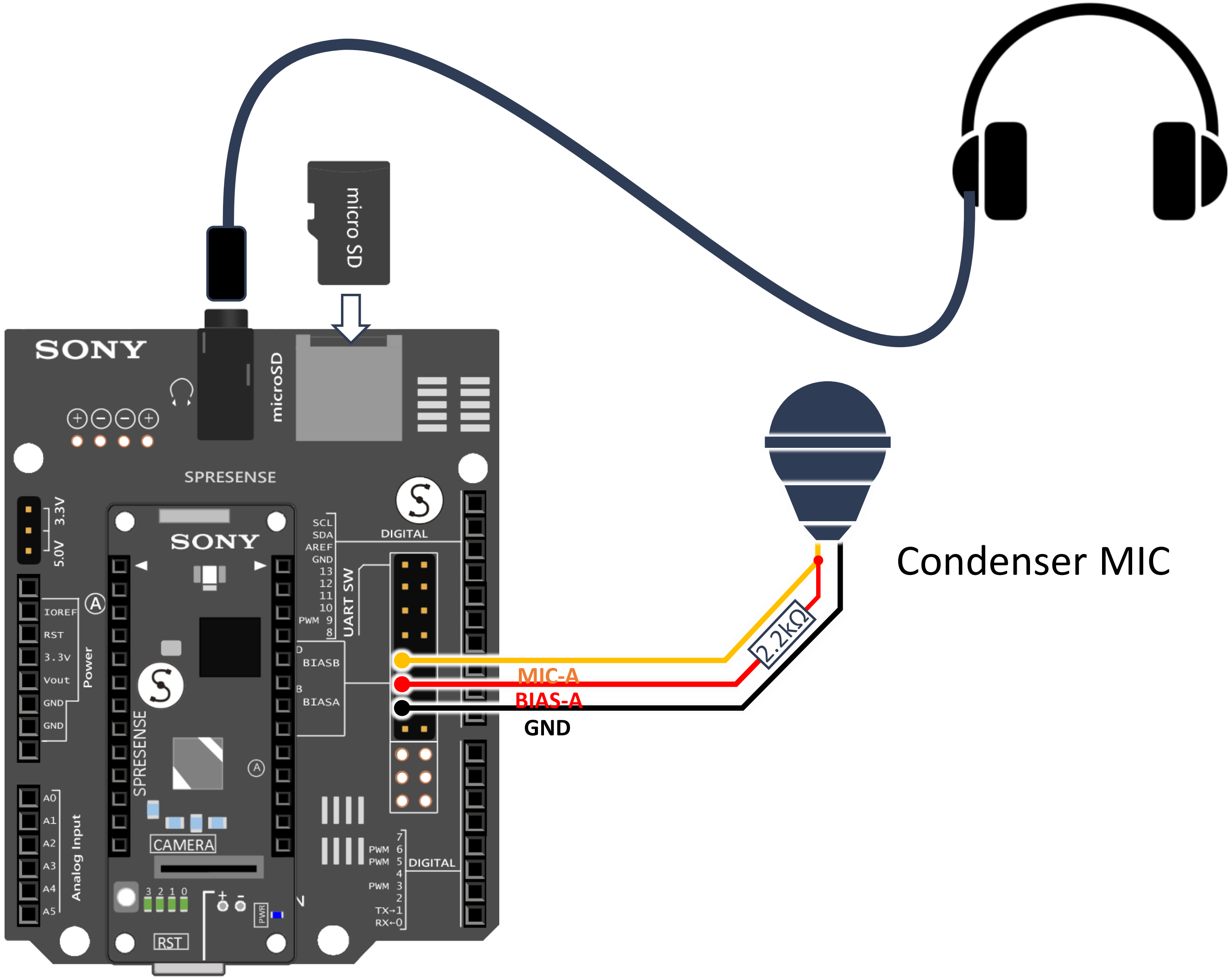
Recording FFT-processed sound

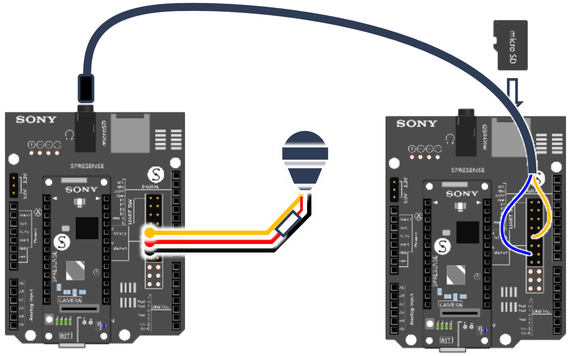
Hardware Configuration

Recording while playing back sounds processed by FFT

SD card access is very time consuming, so the situations in which it can be used are quite limited.

If the process is not completed in time, record by connecting the headphone output to another SPRESENSE microphone input.





Recording FFT-processed sound

Additional implementation for recording sounds processed by FFT conversion to WAV files (48000 Hz only)

Example: Spresense_rfft_wav_recording.ino

RFFT and WAV header settings

```
#include <audio/utilities/wav_containerformat.h>
#include <audio/utilities/wav_containerformat_parser.h>
#define WAV_FILE "test.wav“
WAVHEADER wav_format;
SDClass SD;
File myFile;
uint32_t data_size = 0;
bool b_recording = true;
…
arm_rfft_fast_instance_f32 S;
…
void setup() {
….
while(!SD.begin()){ Serial.println("Insert SD Card");};          SD Card Setting
 if (SD.exists(WAV_FILE)) SD.remove(WAV_FILE);
 myFile = SD.open(WAV_FILE, FILE_WRITE);
```

```
// Write WAV header                                               WAV Header Setting
wav_format.riff    = CHUNKID_RIFF;
wav_format.wave    = FORMAT_WAVE;
wav_format.fmt     = SUBCHUNKID_FMT;
wav_format.fmt_size = FMT_CHUNK_SIZE;
wav_format.format  = FORMAT_ID_PCM;
wav_format.channel = channel_num;
wav_format.rate    = AS_SAMPLINGRATE_48000;
wav_format.avgbyte = AS_SAMPLINGRATE_48000 * channel_num * (bit_length / 8);
wav_format.block   = channel_num * (bit_length / 8);
wav_format.bit     = bit_length;
wav_format.data    = SUBCHUNKID_DATA;
wav_format.total_size = data_size + sizeof(WAVHEADER) - 8;
wav_format.data_size = data_size;
int ret = myFile.write((uint8_t*)&wav_format, sizeof(WAVHEADER));
if (ret != sizeof(WAVHEADER)) {
 Serial.println(“Fail to write file(wav header)”);
 myFile.close();   exit(1);
}
 ….
 arm_rfft_fast_init_f32(&S, SAMPLE_SIZE);
 ….
}
```

Recording FFT-processed sound

Example: Spresense_rfft_wav_recording.ino

Implementation of frontend_pcm_cb function

```
static void frontend_pcm_cb(AsPcmDataParam pcm) {
  static const uint32_t recording_time = 10000; // milli sec
  static uint32_t start_time = millis();

  memset(&input[0], 0, frame_size);
  if (!pcm.is_valid) return;
  /* copy the signal to signal_input buffer */
  memcpy(&mono_input[0], pcm.mh.getPa(), pcm.size);

  /* signal processing start */
  static float pTmpn[sample_size];
  static float p1[sample_size];
  static float p2[sample_size];

  arm_q15_to_float(&mono_input[0], &pTmp[0], SAMPLE_SIZE);          FFT process
  arm_rfft_fast_f32(&S, &pTmp[0], &p1[0], 0);
  int shift = 20;
  memcpy(&p2[shift*2], &p1[0], (SAMPLE_SIZE-shift)*2); /* high pitch */
  arm_rfft_fast_f32(&S, &p2[0], &pTmp[0], 1);
  arm_float_to_q15(&pTmp[0], &mono_input[0], SAMPLE_SIZE);

  memset(&stereo_output[0], 0, frame_size*2);
  for (int n = 0; n < SAMPLE_SIZE; ++n) {
    stereo_output[n*2] = stereo_outputn*2+1] = mono_input[n];
  }
  /* Alloc MemHandle */
  AsPcmDataParam pcm_param;
  if (pcm_param.mh.allocSeg(S0_REND_PCM_BUF_POOL, frame_size) != ERR_OK)  return;
```

```
  /* Set PCM parameters */
  pcm_param.is_end = false;                                          Output Process
  pcm_param.identifier = OutputMixer0;
  pcm_param.callback = 0;
  pcm_param.bit_length = bit_length;
  pcm_param.size = SAMPLE_SIZE *sizeof(int16_t)*2;
  pcm_param.sample = SAMPLE_SIZE *2;
  pcm_param.is_valid = true;
  memcpy(pcm_param.mh.getPa(), stereo_output, pcm_param.size);
  theMixer->sendData(OutputMixer0, outputmixer0_send_cb, pcm_param);

 if (b_recording) {                                                  Recording Process
   data_size += frame_size;
   myFile.write((uint8_t*)&input[0], frame_size);
   if ((millis() - start_time) > recording_time) {
     myFile.seek(0);
     wav_format.total_size = data_size + sizeof(WAVHEADER) - 8;
     wav_format.data_size  = data_size;
     myFile.write((uint8_t*)&wav_format, sizeof(WAVHEADER));
     Serial.println("recording finished!");
     myFile.close();
     b_recording = false;
   }
 }
  return;
}
```

Output sound to MIXER before recording

SPRESENSE